ATM Stream Driver Interface Design
The "DLPI++" Strawman

Request for Comments

Dave Singer and Alagu Periyannan
singer@apple.com

January 1995

Advanced Technology Group
Apple Computer, Inc.

Introduction

This document describes the driver message interface for ATM device drivers in the Apple streams environment, OpenTransport. This driver design is under development within the research organization of Apple, the Advanced Technology Group. It is expected that it will form the basis of a low-level design adopted by Apple, but *no representations can currently be made beyond its current status as a research vehicle in ATG*. Some familiarity with TCP/IP, AppleTalk, and the OpenTransport streams environment is assumed. *Your comments on any aspect of this design are welcome and actively solicited.*

ATM networks are physically point-to-point networks, with connection-oriented protocols used for data transmission. This makes them unlike most existing local area network technologies: particularly, connections must be set up before data can be transferred, and there is no local network broadcast model. Our approach therefore has been to take the standard connection-oriented DLPI design and modify it as little as possible.

The ATM "DLPI++" is a *connection-oriented DLPI* as opposed to an Ethernet DLPI which is connectionless. The "DLPI++" does not support direct IEEE 802 binding and hence can not be used directly by any standard network layer protocols like IP or AppleTalk. Instead, higher-level adaptation modules (outside the scope of this document) are used.

This is a low-level driver interface. In particular, ILMI (UME), Q2110 (QSAAL) and Q2931 (signaling) are independent modules implemented **above** this driver, as are Classical IP (RFC1577) support, LAN Emulation support, or other protocols. This driver is therefore below the level which sees UNI-compliant ATM addresses. Instead, it is working at the virtual circuit level; addresses, to this driver, are VCI/VPIs.

This driver design is for packet-oriented data transfer; AALs 3/4 and 5. ATM adaptation layers such as 1 or 2 will require extensions to this specification.

In general a virtual circuit maps to exactly one stream; a stream normally carries traffic for exactly one VC. Data on these streams is carried in M_DATA messages.

The functionality of the ATM DLPI++ is as follows:

- Enable and Disable one or more incoming/outgoing ATM VCs on a stream
- Specify AAL, Traffic Parameters and QOS on the VC
- Send and Receive AAL packets (3/4 and 5)
- Various simple management functions

Future Revisions

Currently we are allowing more than one VC per stream (for the convenience of higher-level software which must manage many, functionally identical, VCs, such as LAN Emulation); *we intend to remove this feature*. If more than one VC is mapped to a stream, DL_UNITDATA messages are used, with the address field containing the VPI/VCI. This feature raises many issues, for example in the management of QoS, since the DL_INFO_ACK and DL_UDQOS_REQ apply to the whole stream and not a particular VC. *Your comments on having multiple VCs per stream are welcome*.

We expect to add the following in future revisions of this specification; generally, it will not be a problem if drivers do not support all these features. *Your comments on these areas are actively sought.*

Packet Filtering by Content:

There are currently no messages to request traffic filtering by the driver or interface hardware. Such filtering may be desirable for applications such as LAN Emulation, where the incoming traffic from the broadcast-unknown-server (BUS) may contain datagrams for unwanted multicast streams or unicast addresses other than this end-station. The higher-level software will do this filtering if the card and driver are unwilling or unable; but we would like to support cards which can perform content-based filtering in hardware.

Support for Higher Level Functions.

We are also investigating using hardware support for higher-level protocols (e.g. TCP checksum support).

Statistics Management

There are currently no messages defined for reporting statistics and other management. We expect to add these in a later revision of this specification.

Resource Reservation

Before signaling is asked to establish a new VC, higher-level modules will attempt to establish local feasibility, by asking the driver to check the parameters of the VC in advance of its establishment (and therefore, before its VCI is known). This message will look very much like a connect request, with no address (it may be a QoS request issued in IDLE state). Typically this is used before signaling proceeds to setup the call; if the signaling succeeds, a connection request will be sent to associate the VCI/VPI with these resources. If the signaling fails, the resources will be released either by sending another reservation request with different parameters, or by the stream being torn down.

Receive-side buffer supply

It is possible that this driver design will be enhanced in future to allow for the higher-level software to supply receive buffers for a given stream/VC.

Messages Supported

All the data structures referred to here are either in dlpi.h or OpenTptAtm.h header files. Some of the ATM related data structures are reproduced in the next section.

The characteristics of VCs (AAL Type, traffic parameters etc.) are currently passed in a simplified, “digested” structure. We are considering changing this to include whole Q2931 information elements, exactly as they might appear on a network. This would be more flexible, and certainly more complete. However, it would also complicate the driver significantly, in the parsing of those structures and their many optional fields etc. *Your comments on such a change are welcome.*

In general, the expected sequence of messages and states is, for a stream carrying a single VC:

Message	Reply	Valid State	Resulting State
[initial state]			DL_UNBOUND
DL_INFO_REQ	DL_INFO_ACK	any	unchanged
DL_BIND_REQ	DL_BIND_ACK	DL_UNBOUND	DL_IDLE
DL_CONNECT_REQ	DL_CONNECT_CON	DL_IDLE	DL_DATAXFER
M_DATA	no reply	DL_DATAXFER	unchanged
DL_DISCONNECT_REQ	DL_OK_ACK	DL_DATAXFER	DL_IDLE
DL_UNBIND	DL_OK_ACK	DL_IDLE	DL_UNBOUND

General Messages:

DL_INFO_REQ
DL_INFO_ACK

The ATM DLPI++ responds to DL_INFO_REQs with DL_INFO_ACKs. Some of the fields of the DL_INFO_ACK must have specific values,

Maximum and minimum size of AAL 3/4 and 5 packets:
    dl_max_sdu
    dl_min_sdu

MAC layer type - there is no type defined for ATM
    dl_mac_type = DL_OTHER

Service Mode — Connection Oriented
    dl_service_mode = DL_CODLS

Address Information
    (the first or only connected virtual circuit)
    dl_addr_length = sizeof(PVC_Address)
    dl_addr_offset = DL_UNKNOWN or offset if address present

Style and Version of DLPI
    (only style 1 is currently supported; attach/detach are unused)
    dl_provider_style = DL_STYLE1
    dl_version   = DL_VERSION_2

Current State of DLPI
    dl_current_state = DL_DATAXFER or
                       DL_IDLE or
                       DL_UNBOUND

Traffic parameters (currently unused by higher-level software):
  The currently set traffic parameters are reported in the QoS field:
    dl_qos_length = sizeof(ATM_Simple_QOS)
    dl_qos_offset = offset of ATM_Simple_QOS
  The range structure reports the NIC capabilities;
    dl_qos_range_length = sizeof(ATM_Simple_QOS)
    dl_qos_range_offset = offset of ATM_Simple_QOS

In the current QoS report, the driver should report currently set conditions, either from the connection establishment or QoS message, or as a result of network-side operation (e.g. ABR).

In the QoS range report the peak_cell_rate01 (CLP0+1) in the forward and backward directions should indicate the maximum achievable on this NIC. The use of other fields is currently undefined. The reporting of NIC capabilities in this area will be improved.

All other fields are ignored and should be set to 0 (or DL_UNKNOWN in the case of an offset field.)

DL_PHYS_ADDR_REQ
DL_PHYS_ADDR_ACK

This pair of primitives is used to probe the driver for the MAC (OUI) address which should be configured into the physical card. The response should include the MAC address. Currently we do not use the set_phys_addr request, so the card should report the same value for both the current and factory physical addresses.

DL_PHYS_ADDR_ACK
    dl_addr_length = 6 (or 0 if unknown)
    dl_addr_offset = offset (or DL_UNKNOWN if no address present)

Connection Management Messages:

DL_BIND_REQ
DL_BIND_ACK

The Bind messages are supported only to fit into the DLPI and XTI semantics. The ATM DLPI++ responds to a DL_BIND_REQ with a DL_BIND_ACK.

All the fields except dl_service_mode are ignored in the DL_BIND_REQ . If dl_service_mode field is not DL_CODLS, a DL_ERROR_ACK is returned with error DL_UNSUPPORTED. The driver must also check the current state and return a DL_ERROR_ACK with error DL_OUTSTATE, if the state is not DL_UNBOUND.

All fields in the DL_BIND_ACK are set to 0 (or DL_UNKNOWN in the case of an offset field.) Once the DL_BIND_ACK is sent, the driver goes into the DL_IDLE state.

DL_CONNECT_REQ
DL_CONNECT_CON

Connection for this driver is a purely local operation (with the interface card); **no remote network messages are sent**; thus there are neither CONNECT_IND or CONNECT_RES messages used in this context.

The DL_CONNECT_REQ message causes an ATM VC to be activated and a DL_CONNECT_CON to be returned. Once the DL_CONNECT_CON is sent, the driver goes into the DL_DATAXFER state. The destination address in the connect message is used to carry a PVC_Address structure. The QoS information is currently used to carry a simplified version of what signaling might negotiate, an ATM_Simple_QOS structure. This contains the AAL type, traffic parameters and QOS specification.

> dl_addr_length = sizeof(PVC_Address)
> dl_addr_offset = offset of PVC_Address
>
> dl_qos_length = sizeof(ATM_Simple_QOS)
> dl_qos_offset = offset of ATM_Simple_QOS

The DL_CONNECT_CON returns the same address as was sent in the DL_CONNECT_REQ, if the VC enable succeeds, and the same VC_params. If the VC enable fails it returns a DL_ERROR_ACK with error DL_BADADDR. If the driver was not in DL_IDLE, then it should return DL_OUTSTATE; and if any system error occurs (e.g. out of memory), indicate DL_SYSERR with ENOMEM.

Unidirectional connections are indicated with a peak_cell_rate01 of zero in one direction or the other.

DL_DISCONNECT_REQ
DL_UNBIND_REQ

DL_DISCONNECT_REQ disconnects all the currently connected VCs on a stream, and changes the state of the stream from DATAXFER to IDLE. Both the DL_DISCONNECT_REQ and DL_UNBIND_REQ return a DL_OK_ACK on success and DL_ERROR_ACK on error. All fields in DL_DISCONNECT_REQ and DL_UNBIND_REQ are ignored.

A DL_DISCONNECT_REQ is accepted only when the driver is in the DL_DATAXFER state and a DL_UNBIND_REQ is accepted only when the driver is in the DL_IDLE state, otherwise a DL_ERROR_ACK is sent indicating DL_OUTSTATE.

DL_OK_ACK
DL_ERROR_ACK

These messages are sent upstream by the driver to indicated the success or failure of an operation. Various error conditions might be indicated, such as DL_UNSUPPORTED, DL_BADPARAM, DL_OUTSTATE, DL_BADPPA, DL_BADADDR, DL_SYSERR with ENOMEM (when an operation can not be completed because of low memory conditions), DL_SYSERR with other specific system errors or DL_SYSERR with -1 (general system error).

DL_UDQOS_REQ

(Currently unused). This message is used to modify the characteristics of an existing VC (presumed singular). The qos_length and qos_offset are formatted as in the DL_CONNECT_REQ above. The driver should reply with a DL_OK_ACK or DL_ERROR_ACK.

Data Transfer Messages:

The driver should accept the incoming data messages and free them once they are no longer needed. Likewise it should somehow allocate (e.g. using allocb or esballoc) messages which contain incoming data. These will eventually be freed by some higher-level module. It is possible and legitimate to use esballoc to get the buffers back when freed; however, there is no guarantee that they will be freed in a timely fashion.

If the driver is unable to accept data on a stream (e.g. the higher-level software is supplying data faster than the agreed rate, or faster than the card can handle), then it should queue the data messages rather than servicing them. This will apply back-pressure up the stream.

If the driver is unable to supply data to a stream because of flow-control, it should (if it can) apply that back-pressure to the VC (e.g. using ABR mechanisms). It may also discard the data or ignore the flow control indication. Generally this issue should be handled by higher-level protocols and so should not arise; if it does, it is probably safer to discard data than to risk exhausting memory.

M_DATA

When there is only one VC enabled on the stream, all AAL packets are transmitted and received in simple M_DATA messages.

Messages to support multiple VCs per stream

If multiple VCs are bound to a stream, then the sequence of primitives would be like this:

Message	Reply	Valid State	Resulting State
[initial state]			DL_UNBOUND
DL_INFO_REQ	DL_INFO_ACK	any	unchanged
DL_BIND_REQ	DL_BIND_ACK	DL_UNBOUND	DL_IDLE
DL_CONNECT_REQ	DL_CONNECT_CON	DL_IDLE	DL_DATAXFER
DL_IOC_SUBS_CONNECT	M_IOCACK	DL_DATAXFER	unchanged
DL_UNITDATA_REQ	no reply	DL_DATAXFER	unchanged
DL_IOC_SUBS_DISCONNECT	M_IOCACK	DL_DATAXFER	unchanged
DL_DISCONNECT_REQ	DL_OK_ACK	DL_DATAXFER	DL_IDLE
DL_UNBIND	DL_OK_ACK	DL_IDLE	DL_UNBOUND

In general, the driver should handle both M_DATA and DL_UNITDATA_REQ coming downstream at any time; if a DL_IOC_SUBSCONNECT is ever used on the stream, then only DL_UNITDATA messages should be used on that stream from then on.

DL_UNITDATA_REQ
DL_UNITDATA_IND

When there is more than one VC enabled on a stream, then the DL_UNITDATA_REQ and DL_UNITDATA_IND messages are used to indicate the VC on which an AAL packet must be sent or the VC on which an AAL packet was received.

The DL_UNITDATA_REQ specifies the VC to transmit on.

```
        dl_dest_addr_length = sizeof(PVC_Address)
        dl_dest_addr_offset = offset of PVC_Address
```

The DL_UNITDATA_IND specifies the VC on which the AAL packet was received. (The source address need not be present; if the driver is confident that this is a bi-directional VC, then it may fill it in.)

```
        dl_dest_addr_length = sizeof(PVC_Address)
        dl_dest_addr_offset = offset of PVC_Address
        dl_src_addr_length = 0 or sizeof(PVC_Address)
        dl_src_addr_offset = DL_UNKNOWN or offset
```

M_IOCTL Messages (extensions to DLPI):
DL_IOC_SUBS_CONNECT
DL_IOC_SUBS_DISCONNECT

These M_IOCTL messages are used to enable and disable additional VCs on a stream. The DL_IOC_SUBS_CONNECT and the DL_IOC_SUBS_DISCONNECT messages have the same information content as a DL_CONNECT_REQ message.

```
typedef struct {
        UInt32          dl_primitive;
        UInt32          dl_addr_length;
        UInt32          dl_addr_offset;
        UInt32          dl_qos_length;
        UInt32          dl_qos_offset;
        UInt32          dl_growth;
} dl_subs_connect_req_t;
```

The dl_subs_connect_req_t structure, the PVC_Address and the ATM_Simple_QOS for the VC to be enabled or disabled are passed to the driver in an M_DATA message that is chained off an M_IOCTL message. The ioc_cmd field (in the iocblk structure) in the M_IOCTL message and the dl_primitive field in the dl_subs_connect_req_t structure should be the same and set to either DL_IOC_SUBS_CONNECT or DL_IOC_SUBS_DISCONNECT.

The reply to the above M_IOCTL message should be an M_IOCACK message with the correct ioc_rval and ioc_error fields. The ioc_rval field must contain 0 or the appropriate DLPI error code (i.e. DL_BADADDR, DL_OUTSTATE, DL_SYSERR) and the ioc_error field must contain 0 or the appropriate system error, if ioc_rval contains DL_SYSERR.

Note that a DL_IOC_SUBS_DISCONNECT disconnects only one VC; note also that it is possible to disconnect every VC on a stream but for it still to be in “connected” (DL_DATAXFER) state, and ready to accept more DL_IOC_SUBS_CONNECT messages. No traffic should flow, of course, in this somewhat unusual (and unlikely) state.

Data Structures

```
/*###########################
 ATM Addressing
 ###########################*/

/* PVC Address structure */

typedef struct pvc_address {
        UInt32          addrtype;
        UInt8           pad;
        UInt8           vpi;
        UInt16          vci;
} PVC_Address;

/* ATM Address types */
enum ATM_ADDR_TYPE {
         ATM_ADDR_PVC_TYPE=11
};

#define kPVCAddressLength sizeof(PVC_Address)

/*###########################
 ATM Traffic Parameters
 ###########################*/

/* ATM AAL types */
enum aal_type {
   aal_null   = 0,
   aal_type_1 = 1,
   aal_type_34 = 3,
   aal_type_5 = 5,
   aal_type_user = 16
};

typedef UInt32 AAL_Type;

/* PVC QOS specification structure */
typedef struct qos_class {
        UInt8           fwd;
        UInt8           bwd;
        UInt16          pad;
} QOS_Class;

enum Qos_classes {
        qos_class0 = 0,
        qos_class_unspecified = 0,
        qos_class1 = 1,
        qos_class2 = 2,
        qos_class3 = 3,
        qos_class4 = 4
};
```

```
/* PVC Traffic descriptor specification structure */

typedef struct traffic_desc {
        UInt32        fwdPeakCellRate0;
        UInt32        bwdPeakCellRate0;
        UInt32        fwdPeakCellRate01;
        UInt32        bwdPeakCellRate01;
        UInt32        fwdSustCellRate0;
        UInt32        bwdSustCellRate0;
        UInt32        fwdSustCellRate01;
        UInt32        bwdSustCellRate01;
        UInt32        fwdMeanBurstSize0;
        UInt32        bwdMeanBurstSize0;
        UInt32        fwdMeanBurstSize01;
        UInt32        bwdMeanBurstSize01;
        UInt8         bstEffortReq;
        UInt8         fwdTagReq;
        UInt8         bwdTagReq;
        UInt8         pad;
} Traffic_Desc;

/*###########################
 ATM VC Related
 ###########################*/

/* ATM PVC connect message data structure */

typedef struct atm_vc_params {
        AAL_Type          aal;
        QOS_Class         qos;
        Traffic_Desc  traffic;
} ATM_VC_Params;

#define       DL_QOS_SIMPLE_ATM           0x0701
#define       DL_QOS_SIMPLE_ATM_RANGE          0x0702

typedef struct atm_simple_qos {
        UInt32                dl_qos_type;
                                                // DL_QOS_SIMPLE_ATM or
                                                // DL_QOS_SIMPLE_ATM_RANGE
        ATM_VC_Params   params;
} ATM_Simple_QOS;

/*###########################
 Multiple VC per stream Related
 ###########################*/

/* ATM PVC connect message data structure */
#define DL_IOC_SUBS_CONNECT               20
#define DL_IOC_SUBS_DISCONNECT                 21
```