Digital ATM Starter Kit for LAN Emulation Developers

Programmer's Reference

Revision: 1.0
Date: October 2, 1995
Authors: Theodore L. Ross
Douglas M. Washabaugh

(c) Copyright. Digital Equipment Corporation, 1995. All Rights Reserved.

Permission is hereby granted to use, copy, modify, or enhance this software freely, as long as the foregoing copyright of Digital Equipment Corporation and this notice are retained on the software. This software may not be distributed or sublicensed for a fee. Digital makes this software available "AS IS" and without warranties of any kind.

Digital Equipment Corporation

Revision History

Revision	Date	Author	Comments
1.0	2-Oct-95	TLR DMW	Initial Release

Table of Contents

1. INTRODUCTION **5**

2. STARTER KIT CONTENTS **5**

3. ARCHITECTURAL OVERVIEW OF THE SOFTWARE **7**

3.1 STATIC CODE STRUCTURE 7
3.1.1 Top Level Interfaces *7*
3.1.1.1 Upper Datalink Interface 7
3.1.1.2 Lower Port Driver Interface 8
3.1.1.3 UME Interface 8
3.1.2 Top Level Components *8*
3.1.2.1 Connection Manager 8
3.1.2.2 LAN Emulation Client 8
3.1.2.3 Mapping Module 8
3.1.2.4 SVC Module 9
3.1.3 LAN Emulation Client Interfaces *9*
3.1.3.1 The Upper Data Link Interface 9
3.1.3.2 The Proxy Interface 9
3.1.4 LAN Emulation Client Components *10*
3.1.4.1 Data Module 10
3.1.4.2 Control Module 10
3.1.4.3 ARP Module 10
3.2 DYNAMIC OBJECT STRUCTURE 10
3.3 EXECUTION BEHAVIOR 11
3.3.1 Packet Transmission *12*
3.3.1.1 Transmission to an Unknown Address 12
3.3.1.2 Transmission to a Resolved Address 13
3.3.2 Packet Reception *14*
3.3.2.1 Reception of a Data Frame 14
3.3.2.2 Reception of an LE-ARP Response Frame 15

4. MUTUAL EXCLUSION REQUIREMENTS **16**

5. THE LINK STATE MACHINE **16**

6. IMPLEMENTING THE OPERATING SYSTEM SERVICES **17**

6.1 UTL_OS MODULE CREATION AND DELETION 17
6.2 TIMER SERVICES 18
6.3 REGISTER ACCESS SERVICES 18
6.4 MEMORY ALLOCATION SERVICES 18
6.5 PRINTING SERVICES 18
6.6 BUFFER HANDLING SERVICES 19

7. IMPLEMENTING THE LOWER PORT DRIVER **19**

7.1 MANAGING BUFFER POOLS 19
7.2 DRIVER INITIALIZATION 19
7.2.1 Establishing Linkage with the Connection Manager *19*
7.2.2 Allocating a Default Buffer Pool *20*
7.3 VC SETUP AND TEARDOWN 20
7.4 PACKET TRANSMISSION 20
7.4.1 Possible Return Codes for Transmit *20*
7.4.2 Internal vs. External Transmit Packets *21*

7.4.3 Prepending the LAN Emulation Header *21*
7.5 PACKET RECEPTION 22
7.6 REPORTING LINK STATUS TO THE CONNECTION MANAGER 22

8. INTERFACING TO THE SIGNALING PROTOCOL STACK 22

8.1 CONNECTING THE TOP OF THE SIGNALING STACK 23
8.2 CONNECTING THE BOTTOM OF THE QSAAL FUNCTION 23
8.3 USING NON-TRILLIUM SIGNALING 23

9. INITIALIZING THE SYSTEM 23

10. CREATING AND MANAGING LAN EMULATION CLIENTS 24

11. USING THE UPPER DATALINK INTERFACE 24

11.1 RECEIVE ADDRESS FILTERS 24
11.1.1 Multicast Enable *25*
11.1.2 Broadcast Enable *25*
11.1.3 Promiscuous Enable *25*
11.1.4 Managing the Multicast Address Table *25*
11.2 RECEPTION OF DATA PACKETS 25
11.3 TRANSMISSION OF DATA PACKETS 25

12. UME SERVICES REQUIRED BY THE LEC 26

12.1 OBTAINING REGISTERED ATM ADDRESSES 26
12.2 OBTAINING THE UNI VERSION 26

FIGURE 1 - STATIC CODE STRUCTURE - TOP LEVEL 7
FIGURE 2 - STATIC CODE STRUCTURE - LAN EMULATION CLIENT 9
FIGURE 3 - DYNAMIC OBJECT STRUCTURE 11
FIGURE 4 - TRANSMISSION TO AN UNKNOWN ADDRESS 12
FIGURE 5 - TRANSMISSION TO A RESOLVED ADDRESS 13
FIGURE 6 - RECEPTION OF A DATA FRAME 14
FIGURE 7 - RECEPTION OF AN LE-ARP RESPONSE FRAME 15

1. Introduction

The *Digital ATM Starter Kit for LAN Emulation Developers* is a collection of software modules that implement ATM connection management and the ATM Forum LAN Emulation Client. This programmer's reference is intended to aid in the process of porting the *Digital ATM Starter Kit* software into a LAN Emulation Client installation.

The programmer's reference is organized as follows:

- Section 2 describes the contents of the *Digital ATM Starter Kit.* It provides a list of source files with a brief explanation of the contents of each.
- Section 3 provides a detailed architectural overview of the *Digital ATM Starter Kit* including static software structure, dynamic object structure, and several execution scenarios. The purpose of this section is to familiarize the reader with how the software works. This understanding is not necessary for porting the code but is often desirable and is therefore supplied.
- Sections 4 through the end of the document describe the external software interfaces and provide specific information as to how to port the software into a working system.

This programmer's reference does not contain actual function prototypes or type definitions. This information is well documented in the C header files. This is a convention that has been followed to ensure that the detailed technical documentation is up-to-date and accurate.

2. Starter Kit Contents

The *Digital ATM Starter Kit* consists of three major software components: The Connection Manager, the Mapping Storage Module, and the LAN Emulation Client Module. Also included are a number of definition include files and utility modules. The utilities are divided into two categories: Operating System Independent and Operating System Specific. Lastly, several files are provided to aid in connecting a signaling stack to the code.

It is strongly recommended that the common files (i.e. those that are not specific to the hardware platform or operating system) be used unmodified. The more modifications that are made to the common files, the more difficult it will be to incorporate any upgrades that may become available in the future.

Table 1 contains a list of C source files that are included in the *Digital ATM Starter Kit.* The filenames shown in italics are the files most likely to need modification when porting the code. Not all italicized files need modification depending on the platform and operating system that the software will run on.

Table 1 - List of Files in ATM Starter Kit

Module	Filename	Description
Connection Manager	cm.h	General management interface to the Connection Manager
	cm_sap.h	Upper NSAP client interface to the Connection Manager
	cm_svc.h	Interface to Top of Signaling Stack
	cm_drv.h	Lower Port Driver interface to the Connection Manager
	cm.c	Implementation of the Connection Manager

Mapping Module	map.h	Mapping module interface
	map.c	Implementation (linear search with reordering). May be changed to another storage/search algorithm.
LAN Emulation Client	lec.h	Interface to LAN Emulation Client module. Includes management calls and upper datalink calls.
	lec_mgt.h	Definitions of managable attributes within the LEC module.
	lec_data.c	Implementation of the LEC data path. Handles transmit and receive including multicast filters.
	lec_ctrl.h	Interface to the LEC control subcomponent.
	lec_ctrl.c	Implementation of the LEC control state machine. Handles all LAN Emulation control protocols.
	lec_arp.h	Interface to the LEC ARP subcomponent.
	lec_arp.c	Implementation of the LEC ARP function. Handles the LE-ARP cache, ARP entry refresh and aging.
	proxy.h	Optional proxy interface for edge devices.
Interfaces to modules that are not supplied.	line_up.h	Interface to the line_up module (not supplied). This interface is needed because the LEC uses one call in the interface to determine the operating UNI version. This function needs to be implemented.
	addr_reg.h	Interface to the address registration module (not supplied). The LEC uses this interface to obtain ATM addresses prior to operation. This interface needs to be implemented.
Signaling Interface	svc.h	Create and Destroy calls for SVC module
	svc.c	Implementation of SVC module
	svc_info.h	Interface to signaling utilities
	svc_info.c	Implementation of signaling utilities
Utilities, etc.	g_types.h	General purpose data type definitions
	codes.h	Return code definitions
	g_endian.h	Macros for endian-independence
	atm.h	ATM specific type definitions
	system.h	Global type definitions (not specific to any particular interface)
	af_lane.h	ATM Forum LAN Emulation constants and type definitions
	utl.h	Interface to OS-independent utilities
	utl.c	Implementation of OS-independent utilities
	utl_os.h	Interface to OS-specific utilities
	utl_os.c	Implementation of OS-specific utilities (the file provided is an example)
	le_disp.h	Interface to display functions

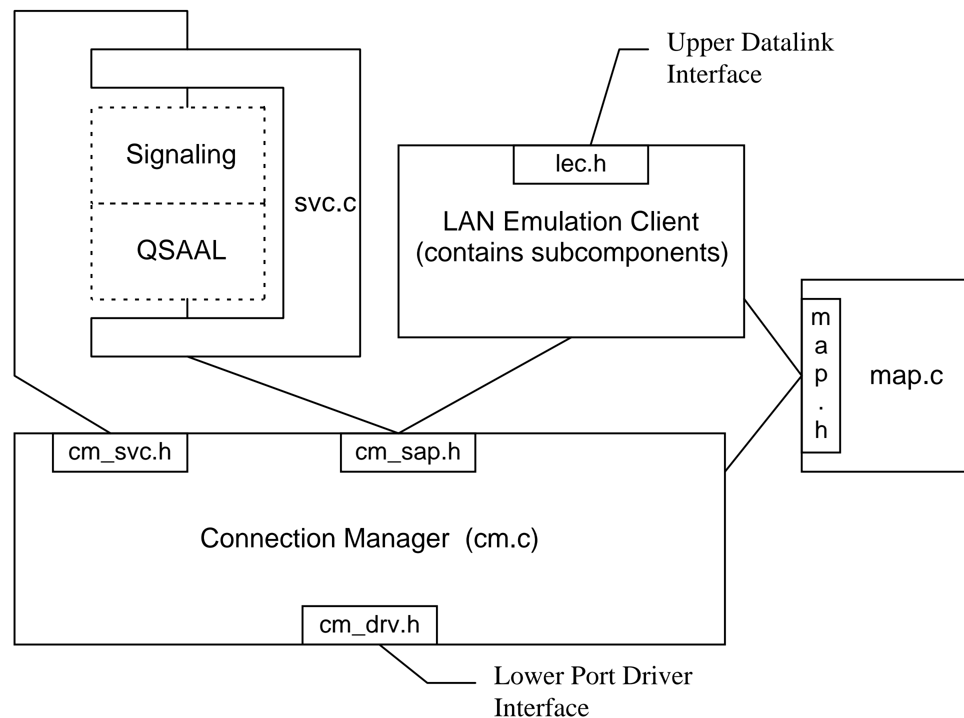
	le_disp.c	Implementation of display functions
	g_event.h	Interface to event reporting module
	g_event.c	Implementation of event reporting module

3. Architectural Overview of the Software

3.1 Static Code Structure

Figure 1 shows the static structure of the *Digital ATM Starter Kit* software. The static structure illustrates how the various modules are connected together. This view does not show how object instances are organized nor does it show how threads of execution flow through the software.

Figure 1 - Static Code Structure - Top Level



3.1.1 Top Level Interfaces

3.1.1.1 Upper Datalink Interface

The upper datalink interface is used to connect the LEC to the bases of protocol stacks at the data link level. This interface is similar to the upper interface in NDIS, ODI, DLPI, and other network device driver standards.

This is a registration style interface that allows multiple logical network interfaces to be created. Each logical interface becomes a separate LEC joined to a separate ELAN.

3.1.1.2 Lower Port Driver Interface

The Lower Port Driver interface connects the Connection Manager to a physical port driver. This is typically a device driver for an AAL5 ATM adapter. The interface provides calls for VC setup and teardown, and transmission and reception of packets.

3.1.1.3 UME Interface

The LAN Emulation Client requires some UME services (i.e. getting the current UNI version and ATM address allocation). The Digital UME modules are not included in the *Digital ATM Starter Kit* but their interface definitions are. This allows an implementer to write the required UME functions to the interface required by the LAN Emulation Client module.

3.1.2 Top Level Components

3.1.2.1 Connection Manager

The Connection Manager plays a central role in the architecture. All packets that flow in and out of the ATM adapter go through the Connection Manager. It is responsible for assigning VC identifiers (VPI and VCI) to transmit packets and for demultiplexing receive packets to the appropriate upper-level components. Receive packet demultiplexing is done on the basis of the VC identifier of the packet.

The Connection Manager provides a connection-oriented interface to Network Service Access Points (NSAPs). Conceptually, NSAPs are the endpoints of VCs. An arbitrary number of NSAP clients may register with the Connection Manager. NSAP clients may request the setup of PVCs and SVCs and may transmit and receive data on those channels.

3.1.2.2 LAN Emulation Client

The LAN Emulation Client module implements the LAN emulation standard as it applies to ATM end-stations. It handles initialization, joining of emulated LANs, the LE-ARP protocol, broadcast and multicast of packets, and normal transmission and reception of unicast packets.

The LAN Emulation Client is based on the ATM Forum standard for LAN Emulation.

3.1.2.3 Mapping Module

The Mapping Module provides a fast lookup data storage service that is used by the Connection Manager and the LAN Emulation Client. The Connection Manager uses the services of this module to keep lists of ATM addresses and VCs. The elements on each list are linked together providing relationships between ATM addresses and VCs.

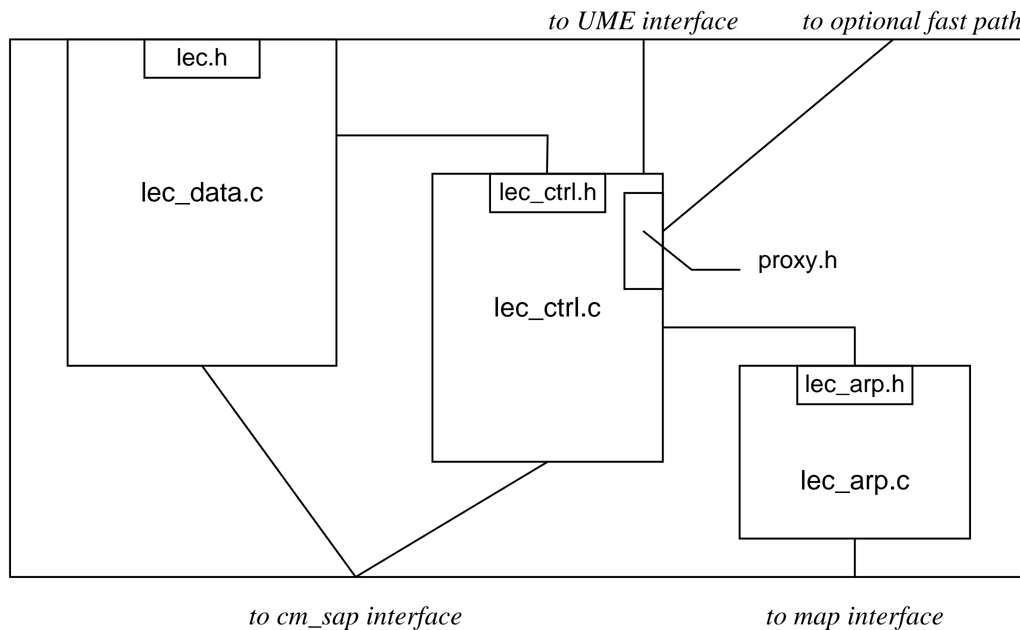
The LAN Emulation Client uses the Mapping Module to maintain a list of MAC addresses for each emulated LAN that the module has joined. The entries in the MAC address list become linked to ATM address entries and VC entries as address resolution and VC setup are achieved respectively. This linkage allows the LAN Emulation Client to determine which VC to use for transmission based on a single MAC address lookup.

3.1.2.4 *SVC Module*

The SVC module provides an object-oriented shim between the Connection Manager and the signaling stack. It connects both the top and bottom of the signaling stack to the Connection Manager.

3.1.3 LAN Emulation Client Interfaces

Figure 2 - Static Code Structure - LAN Emulation Client



3.1.3.1 *The Upper Data Link Interface*

The Upper Data Link Interface is offered by the LAN Emulation Client Module to the upper data link. It is a registration style interface. Before the data link can use the LAN Emulation Client, it must register and in the process, join an emulated LAN. The registration call includes parameters specifying which emulated LAN the driver wishes to join.

3.1.3.2 *The Proxy Interface*

The use of this interface is optional. It is used only in systems that behave as proxy clients (i.e. LAN Emulation bridges and routers). This interface provides access to proxy databases (like bridge address databases) for two purposes:

- The LEC may query the database to see if it should respond to an incoming LE-ARP request;
- The database may contain VC information per address to implement a bridge/router fast-path. This interface allows the LEC to notify the database of changes to MAC address to VC relationships.

The proxy interface is provided so the LAN Emulation Client can function in an edge device with a hardware implementation of bridging/routing. Rather than have the LAN Emulation Client software handle each data packet, the hardware fast-path does so with the information provided

through the proxy interface. In this kind of installation, the LAN Emulation Client module need only handle control packets and transmit packets addressed to unknown unicast addresses.

3.1.4 LAN Emulation Client Components

3.1.4.1 Data Module

The Data module implements the data path through the LAN Emulation Client. It causes a lookup to take place on each transmitted destination address and forwards or discards packets appropriately. It also handles the receive address filters, deciding which received data packets are to be discarded and which are to be passed up to the data link.

3.1.4.2 Control Module

The Control module has the following responsibilities:

- It handles all incoming LAN Emulation control frames;
- It formats and transmits all outgoing LAN Emulation control frames (sometimes on behalf of the ARP module);
- It implements the LAN Emulation Client state machine which sequences all of the activities involved in joining ELANs;
- It forwards destination address lookup requests to the ARP module and provides the Data module with the proper VCC to be used (if any) for packet transmission.

3.1.4.3 ARP Module

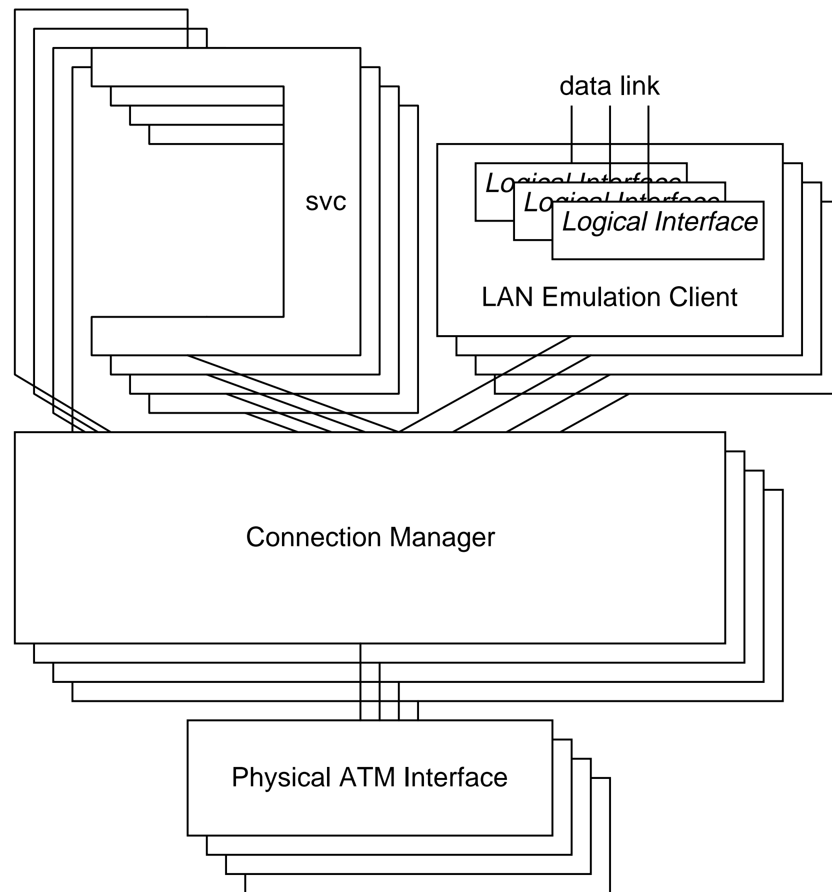
The ARP (Address Resolution Protocol) module implements the LAN Emulation ARP (LEARP) Cache. It has the following responsibilities:

- It tracks the age of each address entry and handles the two-tiered aging mechanism described in the standard;
- It keeps the LEARP entries up-to-date by generating LEARP requests (formatted and transmitted by the Control module);
- It handles incoming LE-ARP responses (parsed by the Control module);
- It calls for the setup of data-direct VCCs;
- It handles the flush protocol;
- It looks up destination MAC addresses for the Control module and decides whether to discard, flood, or directly transmit data frames based on their LEARP entries.

3.2 Dynamic Object Structure

Figure 3 illustrates the dynamic structure of objects in the architecture. In general, there must be a single object of each major component created for each physical ATM port being serviced. For each physical ATM port, there may be many logical Emulated LAN interfaces. Logical interfaces appear to upper layers as separate network interfaces, just as though there were numerous physical Ethernet or Token Ring ports in the system. Note that the mapping module does not appear in Figure 3. This is because the mapping module is not instantiated per physical port. It is an abstract data type which is used by both the Connection Manager and the LAN Emulation Client module. Instances of mapping data structures are allocated as they are needed.

Figure 3 - Dynamic Object Structure



3.3 Execution Behavior

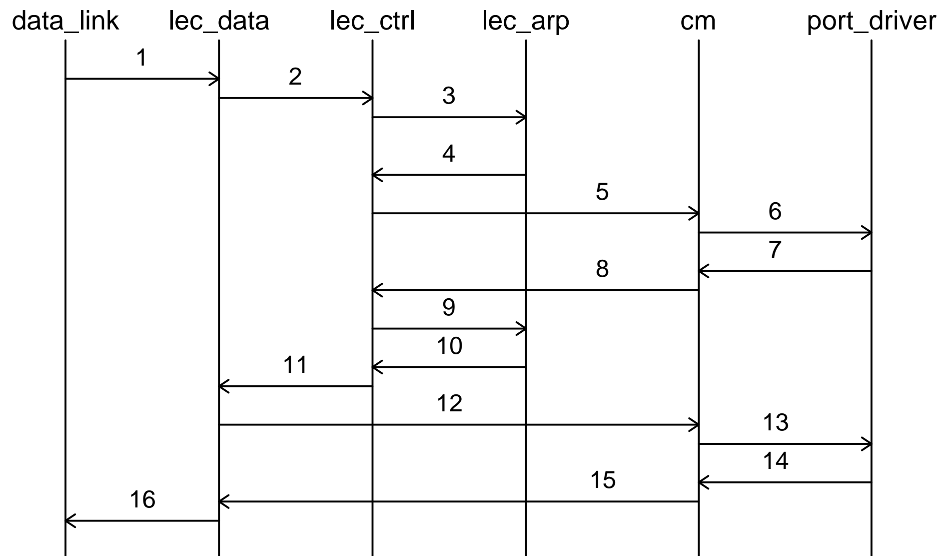
This section describes how threads of execution flow through the ATM modules. The scenarios that are described cover a range of examples to illustrate the operation of the modules.

Note that this is not an illustration of how data packets flow or how protocols operate. It is rather a demonstration of how the various modules interact in some interesting scenarios.

3.3.1 Packet Transmission

3.3.1.1 Transmission to an Unknown Address

Figure 4 - Transmission to an Unknown Address

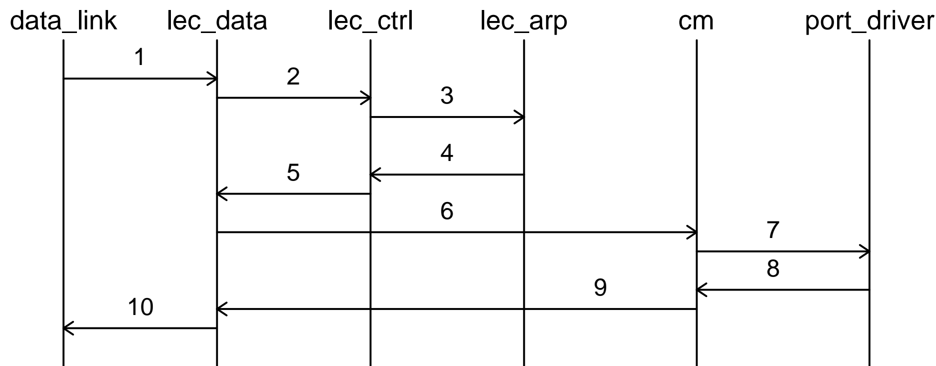


1. The data_link delivers the packet to the LEC for transmission,
2. The lec_data module queries the lec_ctrl module to determine what to do with the packet based on the Destination Address,
3. The lec_ctrl module requests that the lec_arp module lookup the address in the LE-ARP cache.
4. The lec_arp module searches the LE-ARP cache but does not find the address there. It creates a new LE-ARP entry for the address and requests that the lec_ctrl module send an LE-ARP Request frame for that address.
5. The lec_ctrl module builds an LE-ARP Request frame and calls the Connection Manager to request that the frame be sent on the control-direct VCC.
6. The Connection Manager forwards the frame to the Lower Port Driver which copies the frame into a transmit buffer and queues it for transmission.
7. The Lower Port Driver returns success status to the Connection Manager.
8. The Connection Manager returns success status to the lec_ctrl module.
9. The lec_ctrl module returns to the lec_arp module.
10. The lec_arp module returns to the lec_ctrl module summarizing the lookup with a "flood" return command.
11. The lec_ctrl returns to the lec_data module with a return code indicating that the data packet may be forwarded and provides the connection handle for the multicast-send VCC.
12. The lec_data module calls the Connection Manager requesting that the data packet be sent on the VCC provided by the lec_ctrl module.

13. The Connection Manager calls the Lower Port Driver which queues the packet for transmission.
14. The Lower Port Driver returns pending status indicating that the packet is queued but not transmitted.
15. The Connection Manager returns pending status to the lec_data module.
16. The lec_data module returns pending status to the data link.

3.3.1.2 Transmission to a Resolved Address

Figure 5 - Transmission to a Resolved Address

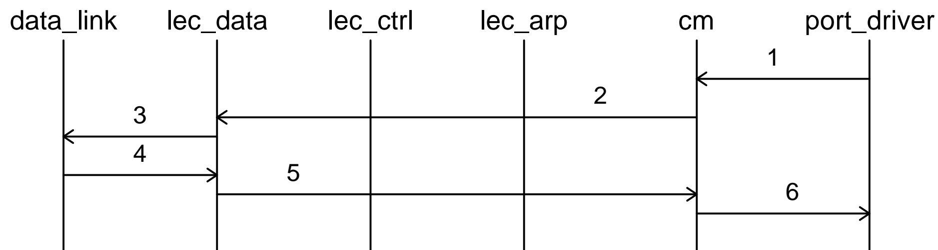


1. The data_link delivers the packet to the LEC for transmission,
2. The lec_data module queries the lec_ctrl module to determine what to do with the packet based on the Destination Address,
3. The lec_ctrl module requests that the lec_arp module lookup the address in the LE-ARP cache.
4. The lec_arp module searches the LE-ARP cache for the address and finds a matching address with an active and current data-direct VCC. It returns a code indicating that the data packet may be forwarded data-direct and provides the connection handle for the data-direct VCC.
5. The lec_ctrl module returns to the lec_data module indicating that the packet may be forwarded and provides the data-direct connection handle.
6. The lec_data module calls the Connection Manager requesting that the data packet be sent on the VCC provided by the lec_ctrl module.
7. The Connection Manager calls the Lower Port Driver which queues the packet for transmission.
8. The Lower Port Driver returns pending status indicating that the packet is queued but not transmitted.
9. The Connection Manager returns pending status to the lec_data module.
10. The lec_data module returns pending status to the data link.

3.3.2 Packet Reception

3.3.2.1 Reception of a Data Frame

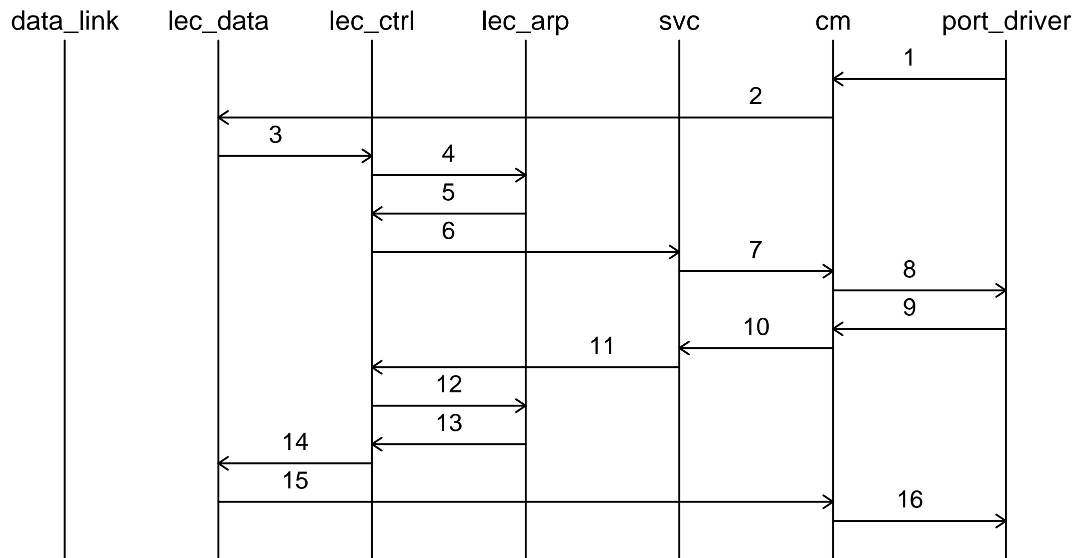
Figure 6 - Reception of a Data Frame



1. The Lower Port Driver receives the packet from the hardware and passes it to the Connection Manager along with the VPI and VCI of the channel on which the packet arrived.
2. The Connection Manager looks up the VPI/VCI and determines that the channel is bound to the lec_data module (SAP client). It calls the lec_data module's receive callback.
3. The lec_data module checks the LEC header to see if the packet is a control packet. It is not so it forwards the packet to the data link.
4. The data link handles the packet and returns to the lec_data module.
5. The lec_data module returns to the Connection Manager.
6. The Connection Manager returns to the Lower Port Driver.

3.3.2.2 *Reception of an LE-ARP Response Frame*

Figure 7 - Reception of an LE-ARP Response Frame



1. The Lower Port Driver receives the packet from the hardware and passes it to the Connection Manager along with the VPI and VCI of the channel on which the packet arrived.
2. The Connection Manager looks up the VPI/VCI and determines that the channel is bound to the lec_data module (SAP client). It calls the lec_data module's receive callback.
3. The lec_data module sees that the packet arrived on a control VCC and forwards it to the lec_ctrl module for handling.
4. The lec_ctrl module decodes the packet and determines it is an LE-ARP response. A call is made to the lec_arp module informing it that an address has been resolved via LE-ARP.
5. The lec_arp module looks up the MAC address in the LE-ARP cache, associates the MAC address with the newly learned ATM address. It also finds that there is no data-direct VCC currently setup to that ATM address. As a result, it calls the lec_ctrl module to request that a data-direct VCC be set up to the ATM address.
6. The lec_ctrl module calls the svc module to request an SVC setup.
7. The svc module handles the connection request, passing it through the signaling stack. The request comes out the bottom of the stack as a SETUP message. The svc module calls the Connection Manager to request that the message be transmitted on the signaling PVC (for which it has a handle).
8. The Connection Manager sends the signaling packet to the Lower Port Driver which copies it into a transmit buffer and queues it.
9. The Lower Port Driver returns success status to the Connection Manager.
10. The Connection Manager returns success status to the svc module.
11. The svc module returns success status to the lec_ctrl module.
12. The lec_ctrl module returns success status to the lec_arp module.
13. The lec_arp module returns to the lec_ctrl module.

14. The lec_ctrl module returns to the lec_data module.
15. The lec_data module returns to the Connection Manager.
16. The Connection Manager returns to the Lower Port Driver.

4. Mutual Exclusion Requirements

The *Digital ATM Starter Kit* code is invoked exclusively by procedure calls. All calls into the code run to completion without blocking. It is assumed that there is at most one thread of execution in the code at any given time per physical ATM port. When there are multiple physical port instances, the data space for each instance is completely separate and there is no re-entrancy problem.

> *For any given physical port instance, it is mandatory that calls into the ATM code not pre-empt one another.*

In message-based installations (like VxWorks) the mutual exclusion requirements are easy to meet. The ATM modules may be collected into a single task with incoming messages being converted into procedure calls. Since messages are handled serially, there is no danger of violating the requirement.

In pre-emptive multitasking environments it may be necessary to use a locking mechanism or raise the interrupt priority level when entering the ATM code.

Timers must also meet the mutual exclusion requirement. Timer expirations result in procedure calls into the ATM code. Timer calls must not pre-empt threads of execution running in the same physical port instance.

5. The Link State Machine

The state of the physical ATM link is maintained by the Connection Manager. The link state changes in response to link events. Link events are reported to the Connection Manager via the `cm_link_event` call (defined in `cm.h`). When the Connection Manager changes the link state, it notifies all of its clients (via the `SAP_LINK_STATUS_CALLBACK` in the `cm_sap.h` interface). The link state and link event values are defined in `system.h`. Refer to Table 2 and Table 3 for descriptions of the link states and link events.

Table 2 - Link State Values

Link State	Description
LINK_DOWN	The physical link is down. Cells are not being received on the ATM interface.
LINK_PHY_UP	The physical link is up. Cells are being received on the ATM interface.
LINK_LINE_UP	The physical link is up and the "line-up" protocol has completed. Line-up consists of a series of ILMI gets to selected remote MIB objects. When line-up is complete, the supported UNI version is known, and the supported VC ranges are known.
LINK_SIG_UP	The physical link is up and the signaling transport is up. Once this state is reached, SVCs may be requested.

Table 3 - Link Event Values

Link Event	Description
LINK_EVENT_PHY_DOWN	The physical ATM interface has stopped receiving cells.
LINK_EVENT_PHY_UP	The physical ATM interface has begun receiving cells.
LINK_EVENT_LINE_UP	The line-up protocol has completed.
LINK_EVENT_SIG_UP	The signaling transport has successfully connected with its UNI peer.
LINK_EVENT_SIG_DOWN	The signaling transport has lost contact with its UNI peer.
LINK_EVENT_RESET_START	The lower port device driver is resetting the physical ATM interface.
LINK_EVENT_RESET_COMPLETE	The lower port device driver has completed resetting the physical ATM interface.

The Connection Manager expects the link states to progress from LINK_DOWN through LINK_SIG_UP (passing through LINK_PHY_UP and LINK_LINE_UP along the way). The changes in the link state cause actions to take place in the ATM system. For example, when the Lower Port Driver detects that the ATM interface is receiving cells, it reports the LINK_EVENT_PHY_UP event to the Connection Manager. This causes a transition from LINK_DOWN to LINK_PHY_UP which alerts the UME (UNI Management Entity, not supplied with the *Digital ATM Starter Kit*) to begin querying the UNI peer for line-up information which is stored for later retrieval. Note that performing the line-up sequence is not mandatory. The UNI version and other information can be manually entered or discovered in some other way. The only thing that is mandatory is that the LINK_EVENT_LINE_UP event be reported to the Connection Manager after the link state transitions to LINK_PHY_UP.

The transition from LINK_PHY_UP to LINK_LINE_UP causes the signaling stack to initialize (now that it knows which UNI version to use). When initialization is successful, it calls the Connection Manager and indicates the LINK_EVENT_SIG_UP event which causes the link state to transition to LINK_SIG_UP.

The LAN Emulation Client instances use the transition to LINK_SIG_UP as their signal to begin attempting to join their respective Emulated LANs.

6. Implementing the Operating System Services

The *Digital ATM Starter Kit* requires that basic operating system services be implemented for the new environment. The interface to the basic OS services is described in the file `utl_os.h`. The file utl_os.c contains an example implementation for Windows NT (running under user mode, not kernel), and should be used as a starting point for your own implementation.

6.1 UTL_OS Module Creation and Deletion

Like many of the other modules in the system, the OS utilities module is object-oriented. Typically, the system creates an instance of the OS utilities for each physical ATM port in the system. This supports environments like Novell Netware that provide separate memory pools for different NICs. There is no reason why a single instance of the OS utilities cannot support multiple ATM physical ports, so this decision is left to the discretion of the designer.

The functions:

- `os_create`, and
- `os_destroy`

must be implemented to provide the capability to create multiple instances of the OS utilities. Refer to the function prototypes and documentation in the interface file (`utl_os.h`) for further details.

6.2 Timer Services

There are two types of timer services provided. The first type is for delaying for an extremely small amount of time (on the order of microseconds). Although the common code does not use this service, device specific code may require.

The second type of timer service is a callback mechanism. This enables the common code to set a timer, and then be called back after a period of time. The implementation should:

- Support a granularity on the order of 100's of milliseconds,
- Support a relatively large number of timers (on the order of 50).

Generally, there are two ways to implement timers. The first way, recommended for implementations that can support a large number of timers, is to create and set a new timer for each one requested. The second way, recommended for implementations that *cannot* support a large number of timers, is to set one master timer, and maintain a queue of entries, each entry representing one timer.

6.3 Register Access Services

The *Digital ATM Starter Kit* code does not use the register access services. These calls appear for the benefit of device driver developers. They need not be implemented to port the *Digital ATM Starter Kit.*

6.4 Memory Allocation Services

Memory allocation services support the allocation and deallocation of "standard" memory. "Standard" memory has only a virtual address, not a physical address. For example, in a NIC implementation, the NIC would not DMA to or from this type of memory. These routines are typically implemented using `malloc` and `free` (or reasonable facsimile thereof).

The functions `os_mem_config` and `os_mem_stats` are not required for normal operation of the software but provide useful testing capability. They can be used to limit the available memory pool and to determine how many bytes have been allocated (peak and current). This feature is most useful in verifying that there are no memory leaks.

6.5 Printing Services

Printing services are provided by a single call (`os_print`). The implementer can have this print to a screen, file, or other destination.

The implementer needs to be aware that printing larger amounts of text to a slow output device can significantly slow down the response time of a driver, which may lead to unforeseen side-effects. To avoid this problem, the implementer may choose to write the information to a memory buffer, and have a separate lower priority process read the memory buffer and print the information.

6.6 Buffer Handling Services

The buffer handling services (`os_buff_hdr_get` and `os_buff_hdr_copy`) provide a mechanism for the software to access the contents of receive buffers (these functions are never used to copy headers from transmit buffers). The common code cannot access these buffers directly because different implementations may have different formats of buffers. For example, in an NDIS 3.0 implementation for an NT device driver, the buffers may be NDIS buffers for transmit buffers, but flat memory for receive buffers.

The `os_buff_hdr_get` function copies the first two octets from the received data packet. This is used on every LAN Emulation packet received so it should execute quickly. The `os_buff_hdr_copy` function copies a variable number of bytes from the front of the packet. This is only used for control packets so it need not be as efficient as `os_buff_hdr_get`.

7. Implementing the Lower Port Driver

The Lower Port Driver interface is specified in the file `cm_drv.h`. Refer to this file for function prototypes and definitions that are needed to implement the Lower Port Driver.

7.1 Managing Buffer Pools

Buffer pools are used by the Lower Port Driver and the Connection Manager to group related VCs together. Transmit channels may need to be grouped by quality of service and/or cell rate to ensure that fast flows don't get stuck behind slow or congested flows. Receive channels are typically grouped by maximum frame size to increase the efficiency of memory buffer usage.

Buffer pools, as used in the *Digital ATM Starter Kit*, are fairly generic and may be used in any one of a number of ways. If the ATM adapter uses transmit and receive queues that are not mapped one-to-one to VCs, buffer pools can be mapped to the queues. If the ATM adapter provides queuing per VC, only a single default transmit buffer pool may be needed.

The designer of the Lower Port Driver decides how buffer pools are to be created and managed. The Connection Manager is informed of the creation and deletion of buffer pools through the `cm_drv_pool_register` and `cm_drv_pool_deregister` functions respectively. Buffer pools may be dynamically registered and deregistered at any time. For proper functioning of the ATM system however, there must always be a default transmit and receive buffer pool registered.

7.2 Driver Initialization

The Connection Manager is the first module to be instantiated during the initialization of the ATM subsystem (refer to section 9 for details on system initialization). The Lower Port Driver is the next module to be initialized. In so doing, it must establish linkage with the Connection Manager instance which is associated with the physical ATM interface served by the driver. The driver must also allocate at least a default pair of buffer pools to be used by the Connection Manager in setting up VCs.

7.2.1 Establishing Linkage with the Connection Manager

Since the Connection Manager instance is created first, its handle (the cm_handle) is known during lower driver initialization. The Connection Manager must be given the handle for the Lower Port Driver before it can make any calls into the driver. This is done using the `cm_drv_config` function.

The Connection Manager must also be provided with the addresses of several calls into the Lower Port Driver. This is accomplished when the driver calls `cm_drv_call_register`.

7.2.2 Allocating a Default Buffer Pool

When the Connection Manager sets up VCs in the driver, it attempts to assign a transmit and receive buffer pool to that VC. Because PVCs are set up very early in the system initialization sequence (i.e. the signaling PVC is set up when the svc module is created), it is important to have at least one default transmit and receive buffer pool available for the early PVCs (signaling, ILMI, and any PVCs used for proprietary value-added features).

7.3 VC Setup and Teardown

The Lower Port Driver is responsible for setting up and tearing down VCs at the request of the Connection Manager. Two of the entry points that the Lower Port Driver registers with the Connection Manager are used for this purpose. They are described in `cm_drv.h` as `DRV_CM_VC_SETUP` and `DRV_CM_VC_TEARDOWN`.

Both the VC setup and teardown operations are considered asynchronous. This is because some ATM adapters may take some time to perform the operations. For example, it may be necessary to flush transmit packets out of a queue before the VC they are destined to can be considered closed. There are two calls into the Connection manager that are used to confirm the completion of VC operations. They are `cm_drv_setup_cfm` and `cm_drv_teardown_cfm` and can be found described in `cm_drv.h`.

Even if VC setup and teardown operations are quick, the confirm functions must be called by the Lower Port Driver before the Connection Manager will consider the VC in question to be setup or torn down. It is acceptable for the `DRV_CM_VC_SETUP` function to call `cm_drv_setup_cfm` before it returns to the Connection Manager (likewise for the VC teardown operation).

When the Connection Manager requests the setup of a VC, it provides the assigned transmit and receive buffer pool contexts and a data structure (called `requested_qos`) describing the VC characteristics. The Lower Port Driver must set up the VC with characteristics that are as close as possible to the requested characteristics. When the setup is confirmed, the Lower Port Driver must indicate to the Connection Manager what the actual VC characteristics are (via the `actual_qos` argument). This is required because hardware limitations may cause the achievable cell rate resolution to differ from the resolution that the Connection Manager assumes. The Connection Manager uses the actual_qos values to keep accurate track of what line resources are used and what is still available.

7.4 Packet Transmission

When the Connection Manager wishes to transmit a packet to the ATM network, it invokes the `DRV_CM_XMT` function in the Lower Port Driver. The call provides a pointer to the packet structure, the length of the packet in bytes, the VPI and VCI of the channel to be used, and the transmit buffer pool context to be used in the transmit. An additional 32-bit argument called `user_data` is provided. The exact structure of the user data is specified in the `system.h` header file.

7.4.1 Possible Return Codes for Transmit

There are four possible status codes that may be returned by the Lower Port Driver's `DRV_CM_XMT` call. They are:

- STATUS_K_SUCCESS,
- STATUS_K_PENDING,
- STATUS_K_CONGESTED, and
- STATUS_K_FAILURE.

STATUS_K_SUCCESS is returned if the packet has been successfully transferred from the buffer that was passed down in the call. This can be a result of a completed programmed-IO operation across a bus or if the packet is copied into another buffer for later transmission. The upper layers will interpret STATUS_K_SUCCESS as confirmation that the buffer has been read and may be overwritten.

STATUS_K_PENDING is returned if the buffer has been successfully queued for transmission. The upper layers will interpret this as an indication that the buffer is still owned by the driver and may not be overwritten. In this case, the Lower Port Driver is obligated to invoke `cm_drv_xmt_done` later when the buffer has been copied. The Connection Manager will route the transmit-done indication up to the original sender of the packet who can in turn put the buffer back in a free pool.

STATUS_K_CONGESTED is returned if the packet cannot be transmitted due to congestion (i.e. full buffers, etc.). In this case, the buffer is returned to the sender intact and the sender has the option of discarding the packet or queuing it for a later retry.

STATUS_K_FAILURE is returned if the packet cannot be transmitted due to some exceptional circumstance (i.e. hardware errors, etc.). The buffer is returned intact to the sender just as with the STATUS_K_CONGESTED status.

7.4.2 Internal vs. External Transmit Packets

The high order bit of the user data argument is called the "internal_source" bit. This bit indicates the source of the packet and gives the Lower Port Driver information about the format of the packet. If the internal_source bit is set, the packet originated from within the *Digital ATM Starter Kit* code. Examples of internally generated packets include signaling packets and LAN Emulation control packets.

Internally generated packets arrive in flat virtual memory space, MUST be copied into a transmit buffer, and the return code resulting from a transmit of an internal packet MUST NOT be STATUS_K_PENDING. The internal code that generates transmit packets will not wait for transmit-done indications. It always assumes that the buffers are writable upon completion of the transmit function.

If the "internal_source" bit in the user data is clear, the packet is externally sourced. This means that the packet came from the upper data link and passed through the LAN Emulation Client and the Connection Manager. In this case, the Lower Port Driver knows what the format of the transmit packet is (because it knows what was provided by the upper data link). Non-internal packets may be copied or queued as-is. If the Lower Port Driver returns STATUS_K_PENDING status (and subsequent transmit-done indications), the upper data link must be able to handle the transmit-done indications.

7.4.3 Prepending the LAN Emulation Header

Because the *Digital ATM Starter Kit* does not know about the format of non-internal packets, it does not prepend the two-octet LAN Emulation header to the outgoing packets. Instead, it supplies

the LAN Emulation header in the user data (in the low-order 16 bits) and relies on the Lower Port Driver to prepend the header to the packet.

IMPORTANT: The LAN Emulation header provided in the user data appears in host byte order. The host-to-network convert routine `hton16` (found in `g_endian.h`) must be used to convert the LAN Emulation header to the proper byte ordering before it is prepended to the outgoing packet.

The "add_le_header" bit in the user data is used to indicate that a header must be prepended to the packet. Note that the "add_le_header" bit is mutually exclusive with the "internal_source" packet. The Lower Port Driver will never be required to prepend a LAN Emulation header to an internally sourced packet.

7.5 Packet Reception

Packet reception is quite straight forward. Once the Lower Port Driver detects that a received packet is in memory, it may call `cm_drv_rcv` to pass that packet up to the ATM subsystem. Along with the received packet buffer, the Lower Port Driver must also provide the VPI/VCI over which the packet arrived, the packet length in bytes, and a user data value. Currently there is no meaning to the user data on received packets so this value may simply be set to zero.

The packet that is passed to the Connection Manager must not contain any ATM Adaptation Layer (AAL) related encapsulation. If the packets are AAL5 encapsulated, the Lower Port Driver must strip off the AAL5 trailer and use the length field from the trailer as the length passed up to the Connection Manager.

7.6 Reporting Link Status to the Connection Manager

The Lower Port Driver must periodically poll the ATM adapter to determine if the hardware is receiving cells. A reasonable poll rate is one per second. If the Lower Port Driver detects a change, it must notify the Connection Manager of that change using the `cm_link_event` call (in `cm.h`).

When the ATM hardware begins receiving cells, the indicated link event should be LINK_EVENT_PHY_UP and when cells stop being received, the event should be LINK_EVENT_PHY_DOWN.

If the Lower Port Driver needs to reset the ATM hardware for any reason, it should issue a LINK_EVENT_RESET_START prior to resetting and a LINK_EVENT_RESET_COMPLETE once reset is complete. This causes the Connection Manager to re-establish all of the PVCs that were open before the reset began.

8. Interfacing to the Signaling Protocol Stack

Digital's ATM code currently uses the signaling stack supplied by Trillium Digital Systems, Inc. Though any signaling stack can be interfaced to the ATM code, it will be easiest to use Trillium's stack.

The interface between the Connection Manager and Signaling is specified in `cm_svc.h`. The glue that connects the signaling stack to the Connection Manager can be found in `svc.c`. The signaling function actually uses two Connection Manager interfaces, the *cm_svc* interface and the *cm_sap* interface. The *cm_svc* interface connects to the upper interface of the signaling stack and handles the signaling requests, indications, responses, and confirmations. The *cm_sap* interface connects to the bottom of the QSAAL layer of the signaling stack. It is used to establish the

signaling PVC (VPI=0, VCI=5) and to transmit and receive QSAAL packets to and from the physical ATM interface.

In installations with more than one physical ATM port, there must be a separate instance of the Connection Manager created for each physical port. Likewise, there must be a separate instance of Signaling created for each physical port. This is because each port has its own VC space and there must be a separate QSAAL transport connection running over each port.

8.1 Connecting the Top of the Signaling Stack

The top of the signaling stack provides services in the form of request, indication, response, and confirmation messages. There are calls that pass between the Connection Manager and the SVC module for each of the messages supported by UNI 3.0 and UNI 3.1. The code in `svc.c` must be provided to glue the signaling interface to the Connection Manager. No code changes are necessary in the Connection Manager.

8.2 Connecting the Bottom of the QSAAL Function

The SVC module registers with the *cm_sap* interface of the Connection Manager and opens up the signaling PVC. It then uses this interface to transmit and receive QSAAL messages going between the signaling stack and the network. This module also handles indicated changes in link state (reported by the Connection Manager). When the link state transitions to LINK_LINE_UP, the signaling stack is to be initialized. The UNI version to be used can be manually configured or determined via ILMI. When the QSAAL connection with the UNI peer has been successfully brought up, the SVC module must signal to the Connection Manager the LINK_EVENT_SIG_UP event (using the `cm_link_event` call).

8.3 Using non-Trillium Signaling

The Trillium dependencies are not widely spread throughout the *Digital ATM Starter Kit* code. If a non-Trillium signaling stack is to be used, the following routines will need to be changed:

- All routines in `svc_info.c`,
- `lc_conn_info_make` in `lec_ctrl.c`, and
- `lc_conn_info_check` in `lec_ctrl.c`.

9. Initializing the System

System initialization is handled independently for each physical ATM adapter being supported. Within each physical instance, the following creation order should be followed:

1. Create the Connection Manager instance,
2. Create the Lower Port Driver instance,
3. Initialize the Lower Port Driver instance,
4. Create the SVC instance,
5. Create the LAN Emulation Client instance,
6. Allow the Lower Port Driver to begin checking to see if cells are being received.

Logical LAN Emulation interfaces may be registered any time after step 5 has completed.

10. Creating and Managing LAN Emulation Clients

The system initialization sequence creates an instance of the LAN Emulation Client module for each physical ATM interface that is present. Simply creating the instance is not sufficient to cause an Emulated LAN to be joined. An ELAN is joined when the layer management function registers with the LEC module (using the `lec_register` call in `lec.h`). There is no architectural limit on how many logical interfaces can be registered on each physical interface. It is simply a matter of how much memory is available to hold the state data structures for each registered ELAN. Furthermore, there is no temporal restriction on when logical interfaces may be registered and deregistered. Logical interfaces may be added and removed dynamically throughout the life of the system.

When `lec_register` is called, the caller supplies all of the salient information about the desired ELAN (i.e. frame type, maximum frame size, initialization method, etc.). This information remains static until the interface is deregistered.

Each registered interface begins the join process when the state of the associated physical link transitions to LINK_SIG_UP. The join process also commences at registration time if the link state is already equal to LINK_SIG_UP.

If the link is lost (i.e. the link state transitions from LINK_SIG_UP) while ELANs are joined, each affected logical interface will become unavailable until the link is restored and the ELANs can be rejoined. A registered logical interface will repeatedly attempt to join an ELAN using the parameters supplied in the `lec_register` call.

11. Using the Upper Datalink Interface

Each logical ELAN interface has associated with it a context (supplied by layer management), an elan_handle (supplied by the LEC module and returned by the `lec_register` function), and three callback functions:

- LEC_EVENT_CALLBACK,
- LEC_RCV_CALLBACK, and
- LEC_XMT_DONE_CALLBACK.

The callback functions are always invoked using the ELAN context supplied at registration. The LEC_EVENT_CALLBACK is used by the LEC module to notify the data link of the status of the logical interface. The interface can simply be available or unavailable. When the ELAN join process successfully completes, the LEC_EVENT_CALLBACK function is called to inform the data link that the interface is available and to provide the negotiated maximum frame size and the name of the joined ELAN. If the registered interface becomes non-operational (due to loss of link or loss of a control VC), the LEC_EVENT_CALLBACK will be invoked to notify the data link that the interface has become unavailable.

11.1 Receive Address Filters

Each registered logical interface has three address filters and a multicast address table. The filters are controlled by enable flags that can be read and modified using the `lec_filters_get` and `lec_filters_set` functions (in `lec.h`) respectively.

11.1.1 Multicast Enable

If the multicast enable flag is set, all packets with multicast destination addresses are received and passed up to the data link. If it is not set, only packets with multicast destination addresses that are found in the multicast address table are received.

11.1.2 Broadcast Enable

If the broadcast enable flag is set, all packets with broadcast destination addresses are received and passed up to the data link. If it is not set, no broadcast packets are received.

11.1.3 Promiscuous Enable

If set, the promiscuous enable flag overrides the multicast and broadcast enable flags. In this case, all received on data-direct and multicast VCCs are passed to the data link packets (except those echoed by the Broadcast and Unknown Server). Promiscuous mode on LAN Emulation is less interesting than on real broadcast LANs (like ethernet). The only extra packets it will receive are unicast packets that arrive on the multicast VCC but are not destined to this data link.

11.1.4 Managing the Multicast Address Table

There are two ways to manage the multicast address table. Addresses may be added and deleted individually (using `lec_mcast_add` and `lec_mcast_delete`) or the whole table may be cleared and loaded in one operation (using `lec_mcast_load`). In both cases, the contents of the table can be retrieved using `lec_mcast_get`.

11.2 Reception of Data Packets

The LEC_RCV_CALLBACK function is called when data packets destined to this data link are received. Received packets either arrived on a data-direct VCC or arrived on a multicast VCC with a destination address that matched the enabled address filters of the logical interface.

A pointer to the received packet buffer is passed by reference to the data link. In other words, the `pp_pkt` argument of the callback is a pointer to a pointer to a buffer. The format of the buffer is simply that which is supplied by the Lower Port Driver. The *Digital ATM Starter Kit* does not place any requirements on the format of receive buffers. If the data link copies or otherwise consumes the received buffer, it may return the same pointer to the caller (the LEC). If the received buffer must be preserved beyond the scope of the receive callback, the data link must either return a fresh buffer or a NULL pointer (if there are no buffers available).

All packets received by the data link have a two-byte LAN Emulation header in the first two bytes of the packet. If the packets are to be passed up to a traditional data link interface (i.e. NDIS, ODI, etc.), the LAN Emulation header must first be stripped off.

11.3 Transmission of Data Packets

Data packets are transmitted by the data link to a logical interface using the `lec_xmt` call. The elan_handle argument indicates which logical interface is to be used to transmit the packet. A pointer to the transmit packet buffer is passed by value (unlike the receive packet buffers) in the `p_pkt` argument. The format of the transmit buffer is simply that which is expected by the Lower Port Driver. The *Digital ATM Starter Kit* software is not concerned with the format of the buffer.

Because the LEC module needs to know the destination address of the transmit packet, the data link must put the address of the DA field of the packet in the `da` argument of `lec_xmt`. This

address must point to a virtually contiguous region of memory that contains the 6-byte destination address of the packet. In the case of 802.5 interfaces, virtually contiguous region must include the packet header from the destination address to the end of the route designator list. If the transmit packet buffer is not virtually contiguous over the required range (which is highly unlikely), the affected part of the packet header must be copied into a virtually contiguous buffer.

Unlike receive buffers which contain LAN Emulation headers, a transmit buffer need not contain a LAN Emulation header. The LAN Emulation header is prepended by the Lower Port Driver (see section 7.4.3 Prepending the LAN Emulation Header).

12. UME Services Required by the LEC

12.1 Obtaining Registered ATM Addresses

Each LAN Emulation Client instance must have an ATM Address with which to create SVCs and register with the LAN Emulation Server. The interface that is used by the LAN Emulation Control module to request ATM Addresses is described in `addr_reg.h`. Since ILMI functionality is not provided with the *Digital ATM Starter Kit*, this interface must be used to connect to whatever address registration function exists in the ATM installation.

12.2 Obtaining the UNI Version

Since The LAN Emulation Client module is responsible for initializing the signaling information elements before requesting the setup of SVCs, it must know the version of UNI (whether 3.0 or 3.1) that is being used on the physical link. This is because the AAL parameters are used differently for the different UNI versions[1].

The UNI version being supported is typically discovered during the line-up phase of link initialization. Since line-up is optional, this information may be manually entered or discovered in some other way.

The LAN Emulation Client will attempt to learn the UNI version prior to joining an Emulated LAN. This will happen only if the link state is LINK_SIG_UP. The call that is invoked from the LAN Emulation is `line_up_uni_version_get`. Its function prototype can be found in `line_up.h`.

Since the line_up module is not supplied with the *Digital ATM Starter Kit*, the use of the line_up_handle in the above call is left to the discretion of the implementer. The line_up_handle that is passed to the `line_up_uni_version_get` function is simply the same handle that was supplied when the LEC module was created.

---

[1] Refer to the LAN Emulation Over ATM Specification, version 1.0, section 3.3.2.7.1.