DLPI Programmer's Guide

Edition 4

B2355-90139
HP 9000 Networking
E0497

Printed in: United States

Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett- Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

HEWLETT-PACKARD COMPANY 3000 Hanover Street Palo Alto, California 94304 U.S.A.

Use of this manual and flexible disk(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Contents

1. Introduction to DLPI

HP DLPI Features 15
Device File Format 16
Header Files 16

The Data Link Layer 17
The Service Interface 17
Modes of Communication 19
DLPI Addressing 21

Promiscuous Mode Clarifications 26

DLPI Services 27
Local Management Services 30
Binding 32
Reserved IEEESAPS/Ethertypes 32
Connection-mode Services 35
Connectionless-mode Services 43
Raw-mode Services 44
An Example 47

2. DLPI Primitives

Local Management Primitives 51
PPA Initialization/De-initialization 51
DL_HP_PPA_REQ 52
DL_HP_PPA_ACK 53
DL_INFO_REQ 55
DL_INFO_ACK 56
DL_ATTACH_REQ 61
DL_DETACH_REQ 62
DL_BIND_REQ 63
DL_BIND_ACK 65
DL_UNBIND_REQ 66

Contents

DL_SUBS_BIND_REQ 67
DL_SUBS_BIND_ACK 69
DL_SUBS_UNBIND_REQ 70
DL_ENABMULTI_REQ 71
DL_DISABMULTI_REQ. 72
DL_PROMISCON_REQ 73
DL_PROMISCOFF_REQ 75
DL_OK_ACK 76
DL_ERROR_ACK 77
Optional Primitives to Perform Essential Management Functions . . 78
DL_PHYS_ADDR_REQ 78
DL_PHYS_ADDR_ACK 80
DL_SET_PHYS_ADDR_REQ. 80
DL_GET_STATISTICS_REQ 82
DL_GET_STATISTICS_ACK 82
DL_HP_MULTICAST_LIST_REQ 83
DL_HP_MULTICAST_LIST_ACK 84

Connectionless-mode Service Primitives 86
DL_UNITDATA_REQ. 86
DL_UNITDATA_IND 88
DL_UDERROR_IND. 89

Raw Mode Service Primitives 91
DL_HP_RAWDATA_REQ. 91
DL_HP_RAWDATA_IND 92

Connection-mode Service Primitives. 94
Connection-Oriented DLPI Extensions 94
DL_HP_INFO_REQ 94
DL_HP_INFO_ACK 95
DL_HP_SET_ACK_TO_REQ 99
DL_HP_SET_P_TO_REQ 100

Contents

DL_HP_SET_REJ_TO_REQ 101
DL_HP_SET_BUSY_TO_REQ. 101
DL_HP_SET_SEND_ACK_TO_REQ. 102
DL_HP_SET_MAX_RETRIES_REQ 103
DL_HP_SET_ACK_THRESH_REQ 104
DL_HP_SET_LOCAL_WIN_REQ 105
DL_HP_SET_REMOTE_WIN_REQ 106
DL_HP_CLEAR_STATS_REQ. 107
DL_HP_SET_LOCAL_BUSY_REQ 108
DL_HP_CLEAR_LOCAL_BUSY_REQ 109
DL_CONNECT_REQ 110
DL_CONNECT_IND 111
DL_CONNECT_RES 113
DL_CONNECT_CON 115
DL_TOKEN_REQ. 117
DL_TOKEN_ACK. 117
DL_DATA_REQ 118
DL_DATA_IND. 119
DL_DISCONNECT_REQ. 119
DL_DISCONNECT_IND 121
DL_RESET_REQ 122
DL_RESET_IND. 123
DL_RESET_RES 124
DL_RESET_CON 125

Primitives to Handle XID and TEST Operations 127
DL_TEST_REQ. 127
DL_TEST_IND 128
DL_TEST_RES 130
DL_TEST_CON 131
DL_XID_REQ 132
DL_XID_IND 133

Contents

DL_XID_RES. 135
DL_XID_CON 136

DLPI States 138

A. Sample Programs

Connection Mode 144

Connectionless Mode 156

Raw Mode 167

Glossary

Printing History

The manual printing date and part number indicate its current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. the manual part number will change when extensive changes are made.

Manual updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. See your HP sales representative for details.

Third Edition: December 1995

Fourth Edition: April 1997

Preface

This guide provides STREAMS kernel-level programming information that is specified by the ISO Data Link Service Definition DIS 8886 and Logical Link Control DIS 8802/2 (LLC). Where the two standards do not conform, DIS 8886 prevails.

This guide assumes familiarity with the OSI Reference Model terminology, OSI Data Link Services, and STREAMS. It is organized as follows:

Chapter 1 **Introduction to DLPI**

This chapter provides an overview of DLPI, including addressing information and information on DLPI services.

Chapter 2 **DLPI Primitives**

This chapter describes local management primitives, connectionless mode primitives, raw mode primitives, and primitives to handle XID and TEST operations.

Appendix A **Sample Programs**

This appendix contains sample programs for connection mode, connectionless mode, and raw mode.

1 Introduction to DLPI

The Data Link Provider Interface (DLPI) is an industry standard definition for message communications to STREAMS-based network interface drivers. A service provider interface is a specified set of messages and the rules that allow passage of these messages across layer boundaries.

HP DLPI Features

Hewlett-Packard's implementation of the Data Link Provider Interface, HP DLPI, conforms to the DLPI Version 2.0 Specification as a Style 2 provider. HP DLPI offers data link service users:

- Clone (maximum 900) and non-clone (maximum 100) access.
- Support for Ethernet/IEEE802.3, FDDI, Fibre Channel, 100VG and Token Ring.
- Support for connectionless and connection-mode services (connection-mode services are supported only over IEEE802.3 and Token Ring).
- Also support for raw-mode services. For details on raw mode, see the DL_BIND_REQ, DL_HP_RAW_REQ and DL_HP_RAW_IND primitives. Raw mode is supported on Ethernet/802.3, FDDI, Token Ring, Fibre Channel and 100VG.
- Style 2.
- I_STR ioctl is supported for doing device-specific control/diagnostic requests.
- Priority messages are supported over 100VG (see DL_UNIT_DATA_REQ primitive).
- For support of third-party devices, refer to the third-party user manuals.
- Support for the following HP products: Ethernet/IEEE802.3, FDDI, Fibre Channel, 100VG and Token Ring.
- The following devices support all levels of promiscuous mode: NIO ethernet LAN, J2146A (HP-PB) NIO LAN only (the 36967A-20N (HP-PB) card is NOT supported), CIO ethernet LAN driver, Series 700 core and HP EISA LAN. For support of third-party devices, refer to the third-party user manuals.

NOTE The HP ATM adapter provides its own "native" DLPI provider, which should not be confused with this DLPI provider.

HP DLPI does not currently include:

- Quality of Service (QOS) management.

- Connection Management STREAMS; DL_SUBS_BIND_REQ and DL_SUBS_UNBIND_REQ over connection-oriented STREAMS.
- Acknowledged connectionless-mode services.

Device File Format

The following is a description of the device file formats required for accessing the STREAMS DLPI LAN driver.

Name	Type	Major #	Minor #	Access
/dev/dlpi	c	72	0x77	Clone access
/dev/dlpiX	c	119	0xX	Non-clone access

NOTE HP DLPI supports up to 100 non-clone device files. HP recommends that device file names follow the naming convention `/dev/dlpiX`, where `X` is the number of the device.

Header Files

There are two DLPI header files: `dlpi.h` and `dlpi_ext.h`. Both are in `/usr/include/sys`. `dlpi.h` contains definitions for the standard DLPI primitives. `dlpi_ext.h` contains definitions for the HP extended DLPI primitives.

The Data Link Layer

The data link layer (layer 2 in the OSI Reference Model) is responsible for the transmission and error-free delivery of bits of information over a physical communications medium. The model defines networking functionality at several layers and service providers between the layers.

A model of the data link layer is presented here to describe concepts that are used throughout this guide. It is described in terms of an interface architecture, as well as addressing concepts needed to identify different components of that architecture. The description of the model assumes familiarity with the OSI Reference Model.

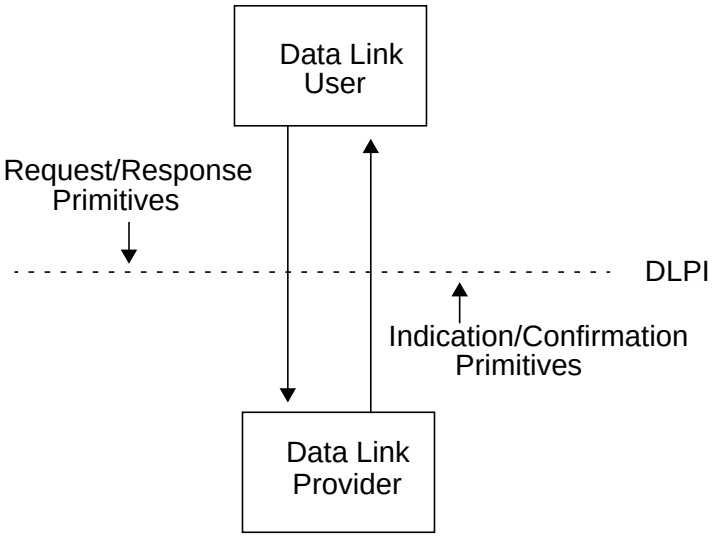
The Service Interface

Each layer of the OSI Reference Model has two standards:

- one that defines the services provided by the layer.
- one that defines the protocol through which layer services are provided.

DLPI is an implementation of the first type of standard. It specifies an interface to the services of the data link layer. Figure 1-1 illustrates how DLPI performs this function.

Figure 1-1 Abstract View of DLPI



The data link interface is the boundary between the network and the data link layers of the OSI Reference Model. The network layer entity is the user of the services of the data link interface (DLS user), and the data link layer entity is the provider of those services (DLS provider). This interface consists of a set of primitives that provide access to the data link layer services, plus the rules for using those primitives (state transition rules). A data link interface service primitive might request a particular service or indicate a pending event.

To provide uniformity among the various UNIX system networking products, an effort is underway to develop service interfaces that map to the OSI Reference Model. A set of kernel-level interfaces, based on the STREAMS development environment, constitute a major portion of this effort. The service primitives that make up these interfaces are defined as STREAMS messages that are transferred between the user and provider of the service. DLPI is one such kernel-level interface, and is targeted for STREAMS protocol modules that either use or provide data link services. In addition, user programs that wish to access a STREAMS-based data link provider directly may do so using the putmsg(2) and getmsg(2) system calls.

Referring to Figure 1-1, the DLS provider is configured as a STREAMS driver, and the DLS user accesses the provider using open(2) to establish a stream to the DLS provider. The stream acts as a communication

endpoint between a DLS user and the DLS provider. After the stream is created, the DLS user and DLS provider communicate via messages discussed later.

DLPI is intended to free data link users from specific knowledge of the characteristics of the data link provider. Specifically, the definition of DLPI hopes to achieve the goal of allowing a DLS user to be implemented independent of a specific communications medium. Any data link provider (supporting any communications medium) that conforms to the DLPI specification may be substituted beneath the DLS user to provide the data link services. Support of a new DLS provider should not require any changes to the implementation of the DLS user.

Modes of Communication

Although DLPI supports three modes of communication, Hewlett-Packard supports connection and connectionless modes. The connection mode is circuit-oriented and enables data to be transferred over a pre-established connection in a sequenced manner. Data may be lost or corrupted in this service mode due to provider-initiated resynchronization or connection aborts.

The connectionless mode is message-oriented and supports data transfer in self-contained units with no logical relationship required between units. Because there is no acknowledgment of each data unit transmission, this service mode can be unreliable in the most general case. However, a specific DLS provider can provide assurance that messages will not be lost, duplicated, or reordered.

Raw mode interface is also supported. Raw mode allows the DLS user to send and receive packets with complete LLC and MAC header information.

Connection-mode Service

The connection-mode service is characterized by four phases of communication:

- Local Management
- Connection Establishment
- Data Transfer
- Connection Release

Local Management. This phase enables a DLS user to initialize a stream for use in communication and establish an identity with the DLS provider.

Connection Establishment. This phase enables two DLS users to establish a data link connection between them to exchange data. One user (the calling DLS user) initiates the connection establishment procedures, while another user (the called DLS user) waits for incoming connect requests. The called DLS user is identified by an address associated with its stream.

A called DLS user may either accept or deny a request for a data link connection. If the request is accepted, a connection is established between the DLS users and they enter the data transfer phase.

For both the calling and called DLS users, only one connection my be established per stream. Thus, the stream is the communication endpoint for a data link connection.

The called DLS user may choose to accept a connection on the stream where it received the connect request, or it may open a new stream to the DLS provider and accept the connection on this new, responding stream. By accepting the connection on a separate stream, the initial stream can be designated as a listening stream through which all connect requests will be processed. As each request arrives, a new stream (communication endpoint) can be opened to handle the connection, enabling subsequent requests to be queued on a single stream until they can be processed.

Data Transfer. In this phase, the DLS users are considered peers and may exchange data simultaneously in both directions over an established data link connection. Either DLS user may send data to its peer DLS user at any time. Data sent by a DLS user is guaranteed to be delivered to the remote user in the order in which it was sent.

Connection Release. This phase enables either the DLS user, or the DLS provider, to break an established connection. The release procedure is considered abortive, so any data that has not reached the destination user when the connection is released may be discarded by the DLS provider.

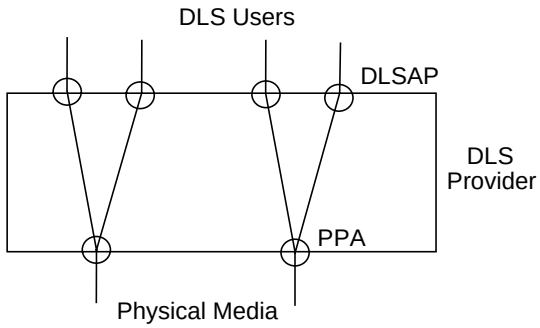
Connectionless-mode Service

The connectionless mode service does not use the connection establishment and release phases of the connection mode service. The local management phase is still required to initialize a stream. Once initialized, however, the connectionless data transfer phase is immediately entered. Because there is no established connection, however, the connectionless data transfer phase requires the DLS user to identify the destination of each data unit to be transferred. The destination DLS user is identified by the address associated with that user.

DLPI Addressing

Each user of DLPI must establish an identity to communicate with other data link users. This identity consists of two pieces. First, the DLS user must somehow identify the physical medium over which it will communicate. This is particularly evident on systems that are attached to multiple physical media. Second, the DLS user must register itself with the DLS provider so that the provider can deliver protocol data units destined for that user. Figure 1-2 illustrates the components of this identification approach, which are explained below.

Figure 1-2 Data Link Addressing Components



Physical Attachment Identification

The physical point of attachment (PPA in Figure 1-2) is the point at which a system attaches itself to a physical communications medium. All communication on that physical medium funnels through the PPA. On systems where a DLS provider supports more than one physical medium, the DLS user must identify which medium it will communicate through.

A PPA is identified by a unique PPA identifier. For media that support physical layer multiplexing of multiple channels over a single physical medium (such as the B and D channels of ISDN), the PPA identifier must identify the specific channel over which communication will occur.

Two styles of DLS provider are defined by DLPI, distinguished by the way they enable a DLS user to choose a particular PPA. The style 1 provider assigns a PPA based on the major/minor device the DLS user opened. This style of provider is appropriate when few PPAs will be supported.

If the number of PPAs a DLS provider will support is large, a style 2 provider implementation is more suitable. The style 2 provider requires a DLS user to explicitly identify the desired PPA using a special attach service primitive. For a style 2 driver, the open(2) creates a stream between the DLS user and DLS provider, and the attach primitive then associates a particular PPA with that stream. The format of the PPA identifier is specific to the DLS provider.

DLPI provides a mechanism to get and/or modify the physical address. The primitives to handle these functions are described in Chapter 2. The physical address value can be modified in a post-attached state. This modifies the value for all streams for that provider for a particular PPA. The physical address cannot be modified if even a single stream for that PPA is in the bound state.

The DLS user uses the supported primitives DL_ATTACH_REQ, DL_BIND_REQ, DL_ENABMULTI_REQ, and DL_PROMISCON_REQ to define a set of enabled physical and SAP address components on a per stream basis. It is invalid for a DLS provider to ever send upstream a data message for which the DLS user on that stream has not requested. The burden is on the provider to enforce the isolation of SAP and physical address space effects on a per-stream basis by any means that it chooses.

HP PPA Format

The PPA number which is passed in the DL_ATTACH_REQ primitive should correspond to the network management ID (NMID) of the interface being attached to. The network management ID is obtainable in one of two ways: 1) the lanscan(1M) command, and 2) programmatically via the HP_PPA_REQ primitive (see Chapter 2).

Data Link User Identification

A data link user's identity is established by associating it with a data link service access point (DLSAP), which is the point through which the user will communicate with the data link provider. A DLSAP is identified by a DLSAP address.

The DLSAP address identifies a particular data link service access point that is associated with a stream (communication endpoint). A bind service primitive enables a DLS user to either choose a specific DLSAP by specifying its address, or to determine the DLSAP associated with a stream by retrieving the bound DLSAP address. The DLSAP address can then be used by other DLS users to access a specific DLS user. The format of the DLSAP address is specific to the DLS provider. However, DLPI provides a mechanism for decomposing the DLSAP address into component pieces. The DL_INFO_ACK primitive returns the length of the SAP component of the DLSAP address, along with the total length of the DLSAP address.

HP's DLSAP Address Format (802.3, Ethernet, Token Ring, FDDI)

Ethernet/IEEE802.3 and FDDI MAC addresses are presented in canonical format. Token Ring MAC addresses are presented in wire format.

DLSAPs are what DLPI defines as an address through which the user will communicate to a Data Link Service (DLS) provider. The content of the DLSAP address will depend on the context in which it is used (i.e. which primitive is being processed or acknowledged). The basic format of the DLSAP address is always the same.

The basic DLSAP address format is:

```
|MAC address | SAP/Ethertype | SNAP (SAP = 0xAA) | [RIF]|
```

[ ] indicates that this information is optional.

The three possible variations of the DLSAP address format based on the protocol value are:

- 802.2 SAP format

  ```
  | DA/SA | DSAP/SSAP | [RIF, up to 18bytes] |
  ```

- Ethertype format

  ```
  | DA/SA | TYPE |
  ```

- SNAP SAP format

  ```
  | DA/SA | 0xAA | SNAP | [RIF, up to 18bytes] |
  ```

HP's DLSAP Address Format for Fibre Channel

The four possible formats for Fibre Channel are:

- 802.2 SAP format

  ```
  | N_Port_Id | Process Associator | FC_Type |DSAP/SSAP|
  ```

- 802.2 SAP without Process Associator format

  ```
  | N_Port_Id | FC_Type | DSAP/SSAP |
  ```

- SNAP/SAP format

  ```
  |N_Port_Id|Process Associator|FC_Type|0xAA|SNAP Info|
  ```

- SNAP/SAP without Process Associator format

  ```
  | N_Port_Id | FC_Type | 0xAA | SNAP Info |
  ```

Certain DLS providers require the capability of binding on multiple DLSAP addresses. This can be achieved through subsequent binding of DLSAP addresses. DLPI supports peer and hierarchical binding of DLSAPs. When the user requests peer addressing, the DLSAP specified in a subsequent bind may be used in lieu of the DLSAP bound in the DL_BIND_REQ. This will allow for a choice to be made between a number of DLSAPs on a stream when determining traffic based on DLSAP values. An example of this would be to specify various ether_type values as DLSAPs. The DL_BIND_REQ, for example, could be issued with an ether_type value of IP, and a subsequent bind could be issued with an ether_type value of ARP. The provider may now multiplex off the ether_type field and allow for either IP or ARP traffic to be sent up this stream.

When the DLS user requests hierarchical binding, the subsequent bind will specify a DLSAP that will be used in addition to the DLSAP bound using a DL_BIND_REQ. This will allow additional information to be specified, that will be used in a header or used for demultiplexing. An example of this would be to use hierarchical bind to specify the Organizational Unique Identifier (OUI) to be used by SNAP.

If a DLS provider supports peer subsequent bind operations, the first SAP that is bound is used as the source SAP when there is ambiguity.

DLPI supports the ability to associate several streams with a single DLSAP, where each stream may be a unique data link connection endpoint. However, not all DLS providers can support such configurations because some DLS providers may have no mechanism beyond the DLSAP address for distinguishing multiple connections. In such cases, the provider will restrict the DLS user to one stream per DLSAP.

Promiscuous Mode Clarifications

The following definitions are being defined for the various levels of promiscuous mode.

DL_PROMISC_PHYS—Before the STREAM has been bound (with the DL_BIND_REQ primitive), the DLPI user receives all traffic on the wire regardless of SAP or address. After the STREAM has been bound, the DLPI user receives all traffic on the wire that matches the protocol(s) the user has bound to on the promiscuous STREAM; this includes protocols bound with the DL_SUBS_BIND_REQ.

DL_PROMISC_SAP—Before the STREAM has been bound (with the DL_BIND_REQ primitive), the DLPI user receives all traffic destined for this interface (physical addresses, broadcast addresses or bound multicast addresses) that matches any SAP enabled on that interface. After the STREAM has been bound, the DLPI user receives only those packets originally destined for the interface that match one of the protocol(s) bound on the promiscuous STREAM.

NOTE The Series 700 core and EISA LAN and100VG drivers are currently the only hardware supporting promiscuous mode which is known to have a MULTICAST_ALL command. This command allows the chip to receive all packets with the group bit set. The other drivers will require that the hardware be in full promiscuous mode and then filter on the group bit in the driver.

DL_PROMISC_MULTI—Before the STREAM has been bound (with the DL_BIND_REQ primitive), the DLPI user receives all multicast packets on the wire regardless of the SAP. After the STREAM has been bound, the DLPI user receives all multicast packets that match one of the protocol(s) bound on the promiscuous STREAM.

NOTE Each LAN interface currently allows only one stream to enable the promiscuous mode service. This restriction will be removed with a future release of the DLPI provider.

DLPI Services

The various features of the DLPI interface are defined in terms of the services provided by the DLS provider and the individual primitives that may flow between the DLS user and DLS provider.

HP DLPI supports two of the three modes of service: connection and connectionless. HP DLPI does not support acknowledged connectionless service. The connection mode is circuit-oriented and enables data to be transferred over an established connection in a sequenced manner. The connectionless mode is message-oriented and supports data transfer in self-contained units with no logical relationship required between units. DLPI also includes a set of local management functions that apply to all modes of service.

DLPI supports the XID and TEST services that appear in the following table. The DLS user can issue an XID or TEST request to the DLS provider. The provider will transmit an XID or TEST frame to the peer DLS provider. On receiving a response, the DLS provider sends a confirmation primitive to the DLS user. On receiving an XID or TEST frame from the peer DLS provider, the local DLS provider sends up an XID or TEST indication primitive to the DLS user. The user must respond with an XID or TEST response frame to the provider.

In addition, raw mode service is now supported. Raw mode allows the DLS user to send and receive packets with complete LLC and MAC headers.

Table 1-1 provides information about the DLPI services that are described in the following sections.

Table 1-1 Cross-Reference of DLS Services and Primitives

Phase of Communication	Service	Primitives
Local Management	Information Reporting	DL_INFO_REQ DL_INFO_ACK DL_ERROR_ACK DL_HP_PPA_REQ DL_HP_PPA_ACK
	Attach	DL_ATTACH_REQ DL_DETACH_REQ DL_OK_ACK DL_ERROR_ACK
	Bind	DL_BIND_REQ DL_BIND_ACK DL_SUBS_BIND_REQ DL_SUBS_BIND_ACK DL_UNBIND_REQ DL_SUBS_UNBIND_REQ DL_OK_ACK DL_ERROR_ACK
	Other	DL_ENABMULTI_REQ DL_DISABMULTI_REQ DL_PROMISCON_REQ DL_PROMISCOFF_REQ DL_OK_ACK DL_ERROR_ACK DL_HP_MULTICAST_LIST_REQ DL_HP_MULTICAST_LIST_ACK

Phase of Communication	Service	Primitives
Connection Establishment	Connection Establishment	DL_CONNECT_REQ DL_CONNECT_IND DL_CONNECT_RES DL_CONNECT_CON DL_DISCONNECT_REQ DL_DISCONNECT_IND DL_TOKEN_REQ DL_TOKEN_ACK DL_OK_ACK DL_ERROR_ACK
Connection-mode Data Transfer	Data Transfer	DL_DATA_REQ DL_DATA_IND
	Reset	DL_RESET_REQ DL_RESET_IND DL_RESET_RES DL_RESET_CON DL_OK_ACK DL_ERROR_ACK
Connection Release	Connection Release	DL_DISCONNECT_REQ DL_DISCONNECT_IND DL_OK_ACK DL_ERROR_ACK
Connectionless-mode Data Transfer	Data Transfer	DL_UNITDATA_REQ DL_UNITDATA_IND
	Error Reporting	DL_UDERROR_IND

Phase of Communication	Service	Primitives
Raw Mode Data Transfer		DL_HP_RAWDATA_REQ DL_HP_RAWDATA_IND
XID and TEST	XID	DL_XID_REQ DL_XID_IND DL_XID_RES DL_XID_CON
	TEST	DL_TEST_REQ DL_TEST_IND DL_TEST_RES DL_TEST_CON

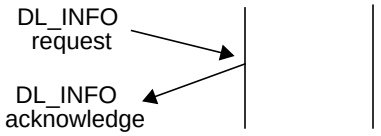
Local Management Services

The local management services apply to the connection and connectionless modes of communication. These services, which fall outside the scope of standards specification, define the method for initializing a stream that is connected to a DLS provider. DLS provider information reporting services are also supported by local management facilities.

Information Reporting Service

This service provides information about the DLPI stream to the DLS user. The message DL_INFO_REQ requests the DLS provider to return operating information about the stream. The DLS provider returns the information in a DL_INFO_ACK message as shown in Figure 1-3.

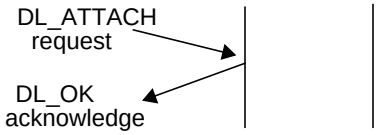
Figure 1-3 Message Flow: Information Reporting



Attach Service

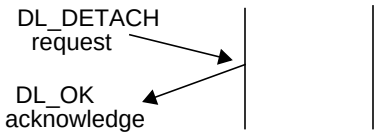
The attach service assigns a physical point of attachment (PPA) to a stream. This service is required for style 2 DLS providers to specify the physical medium over which communications will occur. The DLS provider indicates success with a DL_OK_ACK message and failure with a DL_ERROR_ACK message. The normal message sequence is illustrated in Figure 1-4.

Figure 1-4 Message Flow: Attaching a Stream to a Physical Line



A PPA may be disassociated with a stream using the DL_DETACH_REQ. The normal message sequence is illustrated in Figure 1-5.

Figure 1-5 Message Flow: Detaching a Stream to a Physical Line



Bind Service

The bind service associates a data link service access point (DLSAP) with a stream. The DLSAP is identified by a DLSAP address.

DL_BIND_REQ requests that the DLS provider bind a DLSAP to a stream. It also notifies the DLS provider to make the stream active with respect to the DLSAP for processing connectionless data transfer and connection establishment requests. DL_SUBS_BIND_REQ provides the added capability on binding on multiple DLSAP addresses.

Binding

The following protocol values are currently supported by the DLPI driver:

- IEEE802.2 SAPS
- ethernet types
- SNAP

Valid IEEE802.2 SAPS include even numbers from 0-255, excluding reserved SAPS (see the section “Reserved IEEESAPS/Ethertypes”). Valid ethernet types range from 0x600 to 0xFFFF, excluding reserved ethertypes (see the section “Reserved IEEESAPS/Ethertypes”). The SNAP protocol values contain three bytes of organization ID plus two bytes of additional data. If the first three bytes are 0, the following two bytes are an ethernet type with valid values from 0x0-0xFFFF. If the first three bytes are non-zero, the following two bytes are organization specific with valid values from 0x0-0xFFFF.

IEEE802.2 SAPS and ethernet types are bound to the driver via the DL_BIND_REQ or the DL_SUBS_BIND_REQ (DL_PEER_BIND class only). SNAP protocol values can be logged in two ways. The first method requires you to first bind the SNAP SAP 0xAA via the DL_BIND_REQ primitive. You then must issue a DL_SUBS_BIND_REQ (must be DL_HIERARCHICAL_BIND class) with the five bytes of SNAP data. The second method requires you to bind any non-SNAP protocol value via the DL_BIND_REQ primitive and then issue a DL_SUBS_BIND_REQ (must be DL_PEER_BIND class) with six bytes of data. The first byte must be the SNAP SAP 0xAA followed by five bytes of SNAP data.

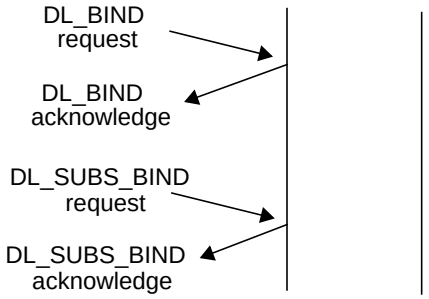
Reserved IEEESAPS/Ethertypes

Refer to the IETF RFC 1010 “Assigned Numbers.”

The DLS provider indicates success with a DL_BIND_ACK or a DL_SUBS_BIND_ACK message and failure with a DL_ERROR_ACK message.

The normal flow of messages is illustrated in Figure 1-6.

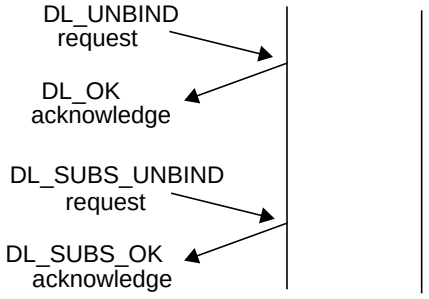
Figure 1-6 Message Flow: Binding a Stream to a DLSAP



DL_UNBIND_REQ requests the DLS provider to unbind all DLSAPs from a stream. The DL_UNBIND_REQ also unbinds all the subsequently bound DLSAPs that have not been unbound. The DLS provider indicates success with a DL_OK_ACK message and failure with a DL_ERROR_ACK message.

DL_SUBS_UNBIND_REQ requests the DLS provider to unbind the subsequently bound DLSAP. The DLS provider indicates success with a DL_OK_ACK message and failure with a DL_ERROR_ACK message, as shown in Figure 1-7.

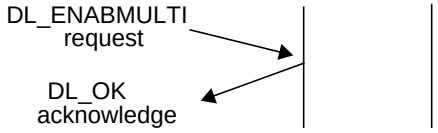
Figure 1-7 Message Flow: Unbinding a Stream from a DLSAP



DL_ENABMULTI_REQ requests the DLS provider to enable specific multicast addresses on a per stream basis. The provider indicates success with a DL_OK_ACK message and failure with a DL_ERROR_ACK message.

The normal message sequence is illustrated in Figure 1-8.

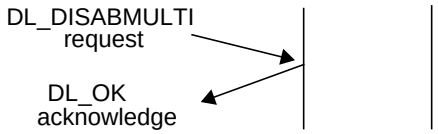
Figure 1-8 **Message Flow: Enabling a Specific Multicast Address on a Stream**



DL_DISABMULTI_REQ requests the DLS provider to disable specific multicast addresses on a per stream basis. The provider indicates success with a DL_OK_ACK message and failure with a DL_ERROR_ACK message.

The normal message sequence is illustrated in Figure 1-9.

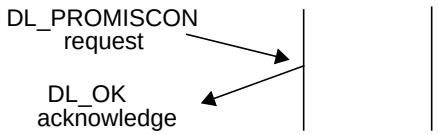
Figure 1-9 **Message Flow: Disabling a Specific Multicast Address on a Stream**



DL_PROMISCON_REQ requests the DLS provider to enable promiscuous mode on a per stream basis, either at the physical level of at the SAP level. The provider indicates success with a DL_OK_ACK message and failure with a DL_ERROR_ACK message.

The normal message sequence is illustrated in Figure 1-10.

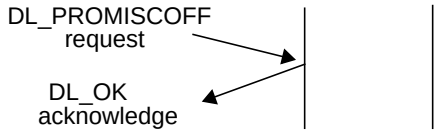
Figure 1-10 **Message Flow: Enabling Promiscuous Mode on a Stream**



DL_PROMISCOFF_REQ requests the DLS provider to disable promiscuous mode on a per stream basis, either at the physical level of at the SAP level. The provider indicates success with a DL_OK_ACK message and failure with a DL_ERROR_ACK message.

The normal message sequence is illustrated in Figure 1-11.

Figure 1-11 Message Flow: Disabling Promiscuous Mode on a Stream



Connection-mode Services

The connection-mode services enable a DLS user to establish a data link connection, transfer data over that connection, reset the link, and release the connection when the conversation has terminated.

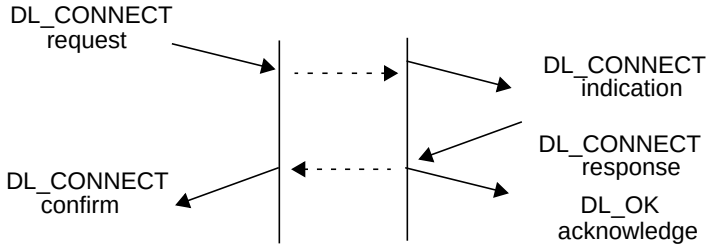
Connection Establishment Service

The connection establishment service establishes a data link connection between a local DLS user and a remote DLS user for the purpose of sending data. Only one data link connection is allowed on each stream.

Normal Connection Establishment. In the connection establishment model, the calling DLS user initiates connection establishment, while the called DLS user waits for incoming requests. DL_CONNECT_REQ requests that the DLS provider establish a connection. DL_CONNECT_IND informs the called DLS user of the request, which may be accepted using DL_CONNECT_RES. DL_CONNECT_CON informs the calling DLS user that the connection has been established.

The normal sequence of messages is illustrated in Figure 1-12.

Figure 1-12 Message Flow: Successful Connection Establishment



Once the connection is established, the DLS users may exchange user data using DL_DATA_REQ and DL_DATA_IND.

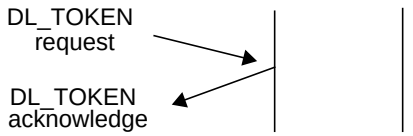
The DLS user may accept an incoming connect request on either the stream where the connect indication arrived or at an alternate, responding stream. The responding stream is indicated by a token in the DL_CONNECT_RES. This token is a value associated with the responding stream and is obtained by issuing a DL_TOKEN_REQ on that stream. The DLS provider responds to this request by generating a token for the stream and returning it to the DLS user in a DL_TOKEN_ACK.

Connection Handoff

Connections may be established on a stream other than that which received the DL_CONNECT_IND by passing a non-zero dl_resp_token in the DL_CONNECT_RES. The dl_resp_token value is obtained by doing a DL_TOKEN_REQ on the stream the connection is being passed to (the data stream). The DL_CONNECT_RES is done on the stream which received the DL_CONNECT_IND (the control stream). Both the control and data streams must be bound on the same local SAP. After the DL_CONNECT_RES, the control stream will be left in the INCON_PEND state if there are more outstanding connect indications; otherwise, it will be left in the IDLE state. The data stream will be in the DATAXFER state.

The normal sequence of messages for obtaining a token is illustrated in Figure 1-13.

Figure 1-13 Message Flow: Token Retrieval



In the typical connection establishment scenario, the called DLS user processes one connect indication at a time, accepting the connection on another stream. Once the user responds to the current connect indication, the next connect indication (if any) can be processed. DLPI also enables the called DLS user to multi-thread incoming connect indications. The user can receive multiple connect indications before responding to any of them. This enables the DLS user to establish priority schemes on incoming connect requests.

Connection Establishment Rejection. In certain situations, the connection establishment request cannot be completed. The following describes the occasions under which DL_DISCONNECT_REQ and DL_DISCONNECT_IND primitives will flow during connection establishment, causing the connect request to be aborted.

Figure 1-14 illustrates the situation where the called DLS user chooses to reject the connect request by issuing DL_DISCONNECT_REQ instead of DL_CONNECT_RES.

Figure 1-14 Message Flow: Called DLS User Rejection of Connection Establishment Attempt

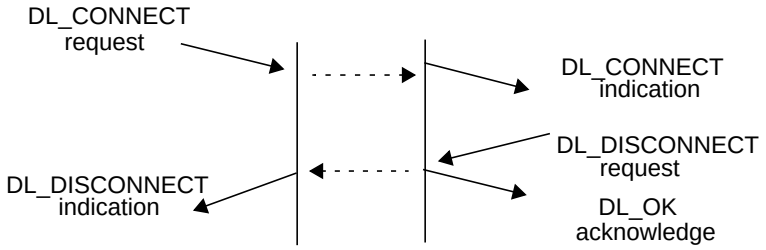


Figure 1-15 illustrates the situation where the DLS provider rejects a connect request for lack of resources or other reasons. The DLS provider sends DL_DISCONNECT_IND in response to DL_CONNECT_REQ.

Figure 1-15 Message Flow: DLS Provider Rejection of a Connection Establishment Attempt

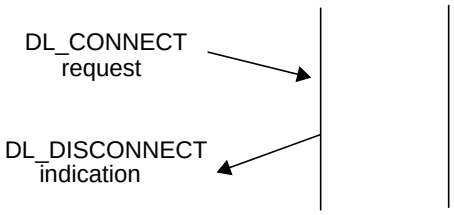


Figure 1-16 through Figure 1-18 illustrate the situation where the calling DLS user chooses to abort a previous connection attempt. The DLS user issues DL_DISCONNECT_REQ at some point following a DL_CONNECT_REQ. The resulting sequence of primitives depends on the relative timing of the primitives involved, as defined in the following time sequence diagrams.

Figure 1-16 Message Flow: Both Primitives are Destroyed by Provider

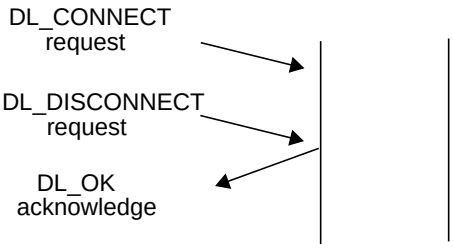


Figure 1-17 Message Flow: DL_DISCONNECT Indication Arrives before DL_CONNECT Response is Sent

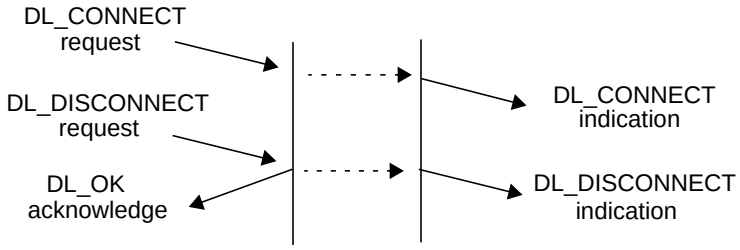
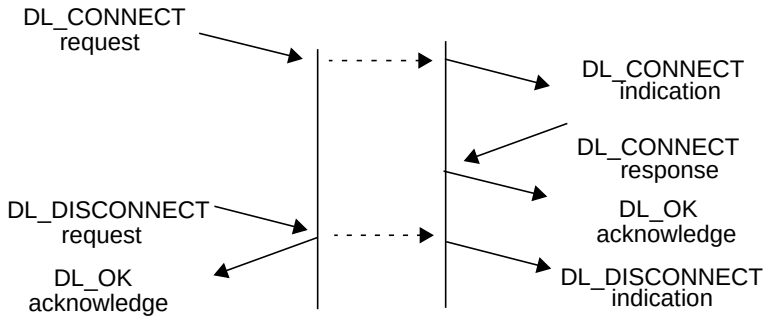


Figure 1-18 **Message Flow: DL_DISCONNECT Indication Arrives after DL_CONNECT Response is Sent**



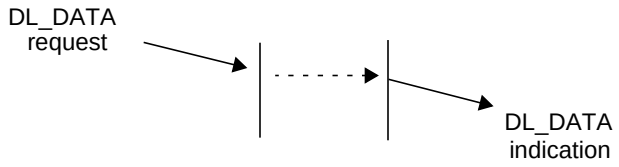
Data Transfer Service

The connection-mode data transfer service provides for the exchange of user data in either direction or in both directions simultaneously between DLS users. Data is transmitted in logical groups called data link service data units (DLSDUs). The DLS provider preserves both the sequence and boundaries of DLSDUs as they are transmitted.

Normal data transfer is neither acknowledged nor confirmed. It is up to the DLS users, if they so choose, to implement a confirmation protocol.

Each DL_DATA_REQ primitive conveys a DLSDU from the local DLS user to the DLS provider. Similarly, each DL_DATA_IND primitive conveys a DLSDU from the DLS provider to the remote DLS user. The normal flow of messages is illustrated in Figure 1-19.

Figure 1-19 **Message Flow: Normal Data Transfer**



Connection Release Service

The connection release service provides for the DLS users or the DLS provider to initiate the connection release. Connection release is an abortive operation and any data in transit (has not been delivered to the DLS user) may be discarded.

DL_DISCONNECT_REQ requests that a connection be released. DL_DISCONNECT_IND informs the DLS user that a connection has been released. Normally, one DLS user requests disconnection and the DLS provider issues an indication of the ensuing release to the other DLS user, as illustrated by the message flow in Figure 1-20.

Figure 1-20 Message Flow: DLS User-Invoked Connection Release

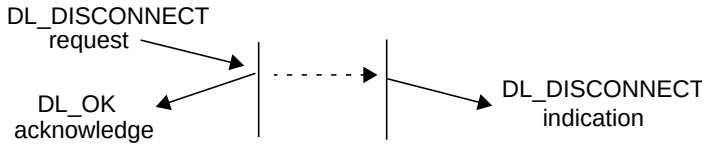


Figure 1-21 illustrates that when two DLS users independently invoke the connection release service, neither received a DL_DISCONNECT_IND.

Figure 1-21 Message Flow: Simultaneous DLS User Invoked Connection Release

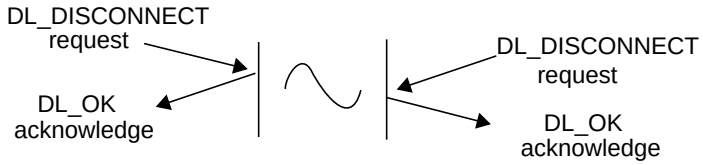
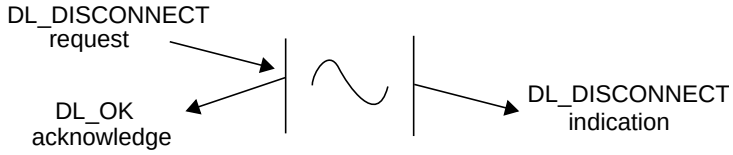


Figure 1-22 illustrates that when the DLS provider and the local DLS user simultaneously invoke the connection release service, the remote DLS user receives a DL_DISCONNECT_IND.

Figure 1-22 Message Flow: Simultaneous DLS User & DLS Provider Invoked Connection Release



Reset Service

The reset service may be used by the DLS user to resynchronize the use of a data link connection, or by the DLS provider to report detected loss of data unrecoverable within the data link service.

Invocations of the reset service will unblock the flow of DLSDUs if the data link connection id congested; DLSDUs may be discarded by the DLS provider. The DLS user or users that did not invoke the reset will be notified that a reset has occurred. A reset may require a recovery procedure to be performed by the DLS users.

The interaction between each DLS user and the DLS provider will be one of the following:

- a DL_RESET_REQ from the DLS user, followed by a DL_RESET_CON from the DLS provider,
- a DL_RESET_IND from the DLS provider, followed by a DL_RESET_RES from the DLS user.

The DL_RESET_REQ acts as a synchronization mark in the stream of DLSDUs that are transmitted by the issuing DLS user; the DL_RESET_IND acts as a synchronization mark in the stream of DLSDUs that are received by the peer DLS user. Similarly, the DL_RESET_RES acts as a synchronization mark in the stream of DLSDUs that are transmitted by the responding DLS user; the DL_RESET_CON acts as a synchronization mark in the stream of DLSDUs that are received by the DLS user which originally issued the reset.

The resynchronizing properties of the reset service are:

- No DLSDU transmitted by the DLS user before the synchronization mark in that transmitted stream will be delivered to the other DLS user after the synchronization mark in that received stream.

- The DLS provider will discard all DLSDUs submitted before the issuing of the DL_RESET_REQ that have not been delivered to the peer DLS user when the DLS provider issues the DL_RESET_IND.
- The DLS provider will discard all DLSDUs submitted before the issuing of the DL_RESET_RES that have not been delivered to the initiator of the DL_RESET_REQ when the DLS provider issues the DL_RESET_CON.
- No DLSDU transmitted by a DLS user after the synchronization mark in that transmitted stream will be delivered to the other DLS user before the synchronization mark in that received stream.

The complete message flow depends on the origin of the reset, which may be the DLS provider or either DLS user. Figure 1-23 illustrates the message flow for a reset invoked by one DLS user.

Figure 1-23 Message Flow: DLS User-Invoked Connection Reset

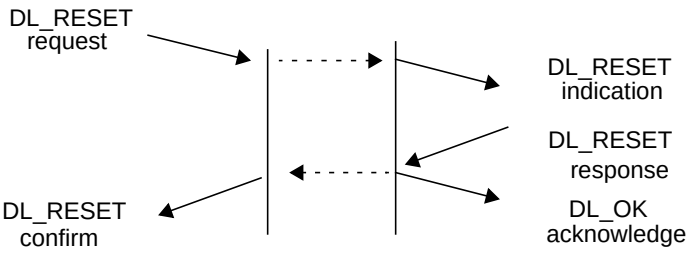


Figure 1-24 illustrates the message flow for a reset invoked by both DLS users simultaneously.

Figure 1-24 Message Flow: Simultaneous DLS User-Invoked Connection Reset

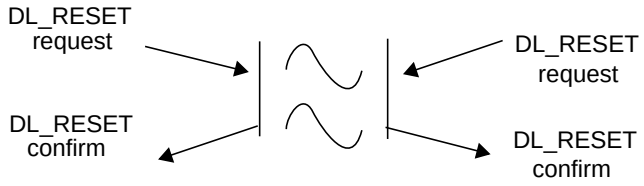


Figure 1-25 illustrates the message flow for a reset invoked by the DLS provider.

Figure 1-25 Message Flow: DLS Provider-Invoked Connection Reset

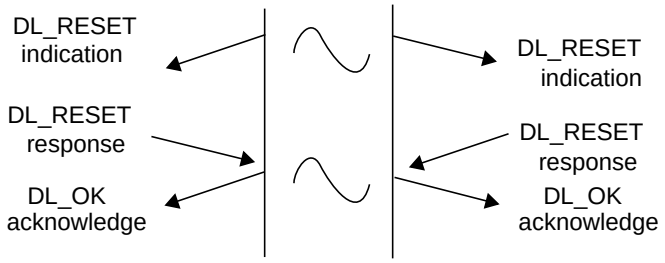
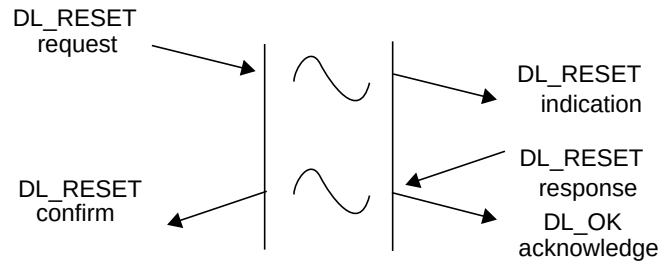


Figure 1-26 illustrates the message flow for a reset invoked simultaneously by one DLS user and the DLS provider.

Figure 1-26 Message Flow: Simultaneous DLS User & DLS Provider-Invoked Connection Reset



Connectionless-mode Services

The connectionless-mode services enable a DLS user to transfer units of data to peer DLS users without incurring the overhead of establishing and releasing a connection. The connectionless service does not, however, guarantee reliable delivery of data units between peer DLS users (e.g. lack of flow control may cause buffer resource shortages that result in data being discarded).

Once a stream has been initialized via the local management services, it may be used to send and retrieve connectionless data units.

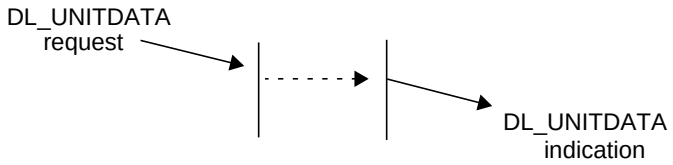
Connectionless Data Transfer

The connectionless data transfer service provides for the exchange of user data (DLSDUs) in either direction or in both directions simultaneously without having to establish a data link connection. Data

transfer is neither acknowledged nor confirmed, and there is no end-to-end flow control provided. As such, the connectionless data transfer service cannot guarantee reliable delivery of data. However a specific DLS provider can provide assurance that messages will not be lost, duplicated, or reordered.

DL_UNITDATA_REQ conveys one DLSDU to the DLS provider. DL_UNITDATA_IND conveys one DLSDU to the DLS user. The normal flow of messages is illustrated in Figure 1-27.

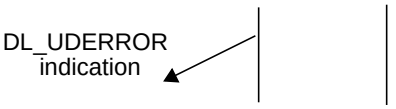
Figure 1-27 Message Flow: Connectionless Data Transfer



Error Reporting Service

The connectionless-mode error reporting service may be used to notify a DLS user that a previously sent data unit either produced an error or could not be delivered. This service does not, however, guarantee that an error indication will be issued for every undeliverable data unit.

Figure 1-28 Connectionless-Mode Error Reporting



Raw-mode Services

The raw-mode services enable a DLS user to transfer packets containing complete MAC and LLC headers to a peer DLS user. The raw-mode service does not guarantee reliable delivery of data units between peer DLS users (e.g. lack of flow control may cause buffer resource shortages that result in data being discarded).

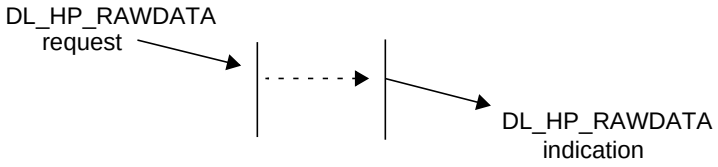
The DLS user requests the raw-mode services by setting the service mode in the DL_BIND_REQ to DL_HP_RAWDLS.

Raw-mode Data Transfer

The raw-mode data transfer service provides the same service as the connectionless data transfer service. The only difference is that the raw-mode DLS user builds the complete MAC and LLC headers prior to data transfer, whereas the connectionless-mode DLS user merely specifies the peer DLS user and the DLS provider then builds the complete MAC and LLC headers before transferring the packet.

The DL_HP_RAWDATA_REQ conveys one DLSDU to the DLS provider. The DL_HP_RAWDATA_IND conveys one DLSDU to the DLS user. The normal flow of messages is illustrated in Figure 1-29.

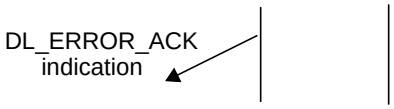
Figure 1-29 Message Flow: Raw Data Transfer



Error Reporting Service

The raw-mode error reporting service provides the same services as the connectionless-mode error reporting services. However, the DL_ERROR_ACK primitive is used in place of the DL_UDERROR primitive to report all error conditions in raw-mode.

Figure 1-30 Raw-Mode Error Reporting



XID and TEST Service

The XID and TEST service enables the DLS user to issue an XID or TEST request to the DLS provider. On receiving a response for the XID or TEST frame transmitted to the peer DLS provider, the DLS provider sends up an XIS or TEST confirmation primitive to the DLS user. On receiving an XID or TEST frame from the peer DLS provider, the local

DLS provider sends up an XID or TEST indication respectively to the DLS user. The DLS user must respond with an XID or TEST response primitive.

If the DLS user requested automatic handling of the XID or TEST response, at bind time, the DLS provider will send up an error acknowledgment on receiving an XID or TEST request. Also, no indications will be generated to the DLS user on receiving XID or TEST frames from the remote side.

XID and TEST Packet Handling

XID and TEST packets are handled differently on connection oriented streams than they are on connectionless streams. On connectionless streams, XID and TEST packets may be sent and received by any stream at any time after binding. On connection oriented streams, XID and TEST packets may be sent and received at any time after binding by streams specifying a non- zero dl_max_conind in the DL_BIND_REQ. Connection oriented streams which specify a zero dl_max_conind in the DL_BIND_REQ will only receive XID and TEST packets after a connection has been established.

LLC Type 2 monitors XID packets sent and received on connection oriented streams. If the stream has a connection established, LLC Type 2 will set the local and remote receive window sizes to those specified in the XID packets.

The normal flow of message is illustrated in Figure 1-31 and Figure 1-32.

Figure 1-31 Message Flow: XID Service

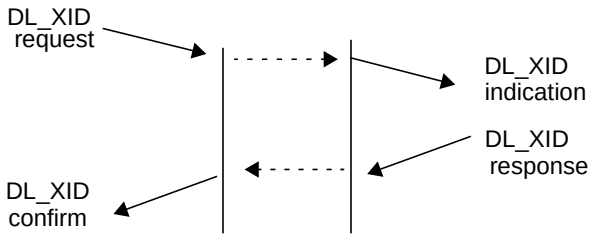
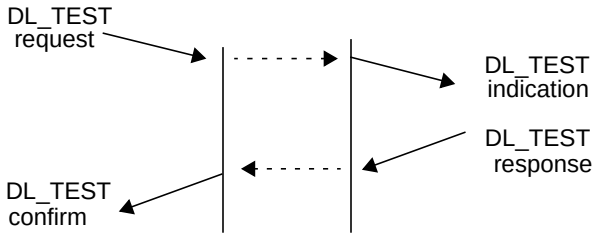


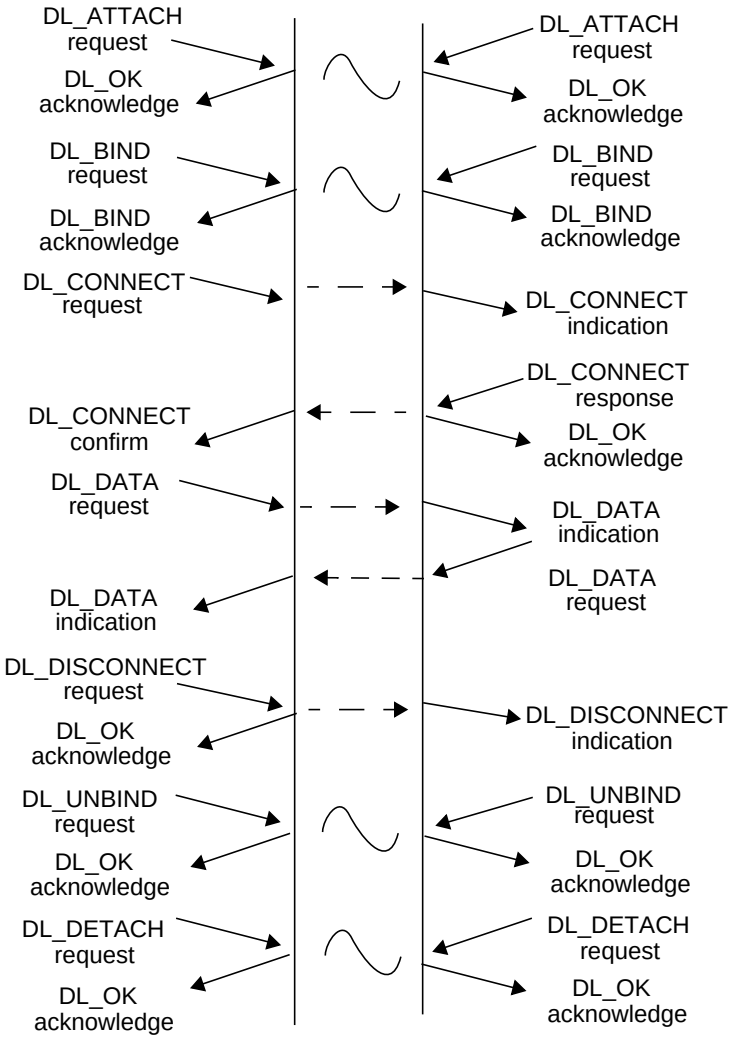
Figure 1-32 Message Flow: Test Service



An Example

To summarize, Figure 1-33 is an example that illustrates the primitives that flow during a complete, connection-mode sequence between stream open and stream close.

Figure 1-33 Message Flow: A Connection-Mode Example



2 DLPI Primitives

The kernel-level interface to the data link layer defines a STREAMS-based message interface between the provider of the data link service (DLS provider) and the consumer of the data link service (DLS user). STREAMS provides the mechanism in which DLPI primitives may be passed between the DLS user and DLS provider.

Before DLPI primitives can be passed between the DLS user and the DLS provider, the DLS user must establish a stream to the DLS provider using open(2). The DLS provider must therefore be configured as a STREAMS driver. When interactions between the DLS user and DLS provider have completed, the stream may be closed.

The STREAMS messages used to transport data link service primitives across the interface have one of the following formats:

- One M_PROTO message block followed by zero or more M_DATA blocks. The M_PROTO message block contains the data link layer service primitive type and all relevant parameters associated with the primitive. The M_DATA block(s) contain any DLS user data that might be associated with the service primitive.
- One M_PCPROTO message block containing the data link layer service primitive type and all relevant parameters associated with the service primitive.
- One or more M_DATA message blocks conveying user data.

The following sections describe the format of the supported primitives. The primitives are grouped into four categories:

- Local Management Service Primitives
- Connectionless-mode Service Primitives
- Connection-mode Service Primitives
- Primitives to handle XID and TEST operations

All of the DLPI extensions listed in this chapter are defined in `<sys/dlpi_ext.h>` and `<sys/dlpi.h>`.

Local Management Primitives

This section describes the local management service primitives. These primitives support the information reporting, Attach and Bind. Once a stream has been opened by a DLS user, these primitives initialize the stream, preparing it for use.

PPA Initialization/De-initialization

The PPA associated with each stream must be initialized before the DLS provider can transfer data over the medium. The initialization and de-initialization of the PPA is a network management issue, but DLPI must address the issue because of the impact such actions will have on a DLS user. More specifically, DLPI requires the DLS provider to initialize the PPA associated with a stream at some point before it completes the processing of the DL_BIND_REQ. Guidelines for initialization and de-initialization of a PPA by a DLS provider are presented here.

A DLS provider may initialize a PPA using the following methods:

- pre-initialized by some network management mechanism before the DL_BIND_REQ is received; or
- automatic initialization on receipt of a DL_BIND_REQ or DL_ATTACH_REQ.

A specific DLS provider may support either of these methods, or possibly some combination of the two, but the method implemented has no impact on the DLS user. From the DLS user's viewpoint, the PPA is guaranteed to be initialized on receipt of a DL_BIND_ACK. For automatic initialization, this implies that the DL_BIND_ACK may not be issued until the initialization has completed.

If pre-initialization has not been performed and/or automatic initialization fails, the DLS provider will fail the DL_BIND_REQ. Two errors, DL_INITFAILED and DL_NOTINIT, may be returned in the DL_ERROR_ACK response to a DL_BIND_REQ if PPA initialization fails. DL_INITFAILED is returned when a DLS provider supports automatic PPA initialization, but the initialization attempt failed. DL_NOTINIT is returned when the DLS provider requires pre-initialization, but the PPA is not initialized before the DL_BIND_REQ is received.

A DLS provider may handle PPA de-initialization using the following methods:

- automatic de-initialization upon receipt of the final DL_DETACH_REQ (for style 2 providers) or DL_UNBIND_REQ (for style 1 providers), or upon closing of the last stream associated with the PPA;
- automatic de-initialization after expiration of a timer following the last DL_DETACH_REQ, DL_UNBIND_REQ, or close as appropriate; or
- no automatic de-initialization; administrative intervention is required to de-initialize the PPA at some point after it is no longer being accessed.

A specific DLS provider may support any of these methods, or possibly some combination of them, but the method implemented has no impact on the DLS user. From the DLS user's viewpoint, the PPA is guaranteed to be initialized and available for transmission until it closes or unbinds the stream associated with the PPA.

DLS provider-specific addendum documentation should describe the method chosen for PPA initialization and de-initialization.

DL_HP_PPA_REQ

This primitive is used to obtain a list of all the valid PPAs currently installed in the system.

This message consists of one M_PCPROTO message block which contains the following structure.

Format

```
typedef struct {
    u_long      dl_primitive;
} dl_hp_ppa_req_t;
```

Parameters

dl_primitive

DL_HP_PPA_REQ

State

The message is valid in any State in which a local acknowledgment is not pending, as described in Appendix B, Allowable Sequence of DLPI Primitives, of the DLPI 2.0 specification.

New State

The resulting state is unchanged.

Response

The DLPI driver responds to this request with a DL_HP_PPA_ACK.

DL_HP_PPA_ACK

This primitive is sent in response to a DL_HP_PPA_REQ; it conveys information on each valid PPA currently installed in the system.

This message consists of one M_PCPROTO message block which contains the following structure and information.

Format

```
typedef struct {
    u_long     dl_primitive;
    u_long     dl_length;
    u_long     dl_count;
    u_long     dl_offset;
} dl_hp_ppa_ack_t;
```

Parameters

dl_primitive

DL_HP_PPA_ACK

dl_length

length of the data area following the DL_HP_PPA_ACK primitive. The data area is formatted as one or more dl_hp_ppa_info_t structures (see below).

dl_count

number of PPAs in the list.

dl_offset

offset from the beginning of the M_PCPROTO block where the dl_hp_ppa_info_t information begins.

```
/* info area in DL_HP_PPA_ACK */
    typedef struct {
        u_long  dl_next_offset;
        u_long  dl_ppa;
        u_char  dl_hw_path[100];
        u_long  dl_mac_type;
        u_char  dl_phys_addr[20];
        u_long  dl_addr_length;
        u_long  dl_mjr_num;
        u_char  dl_name[64]
        u_long  dl_instance_num
        u_long  dl_mtu;
        u_long  dl_hdw_state;
        u_char  dl_module_id_1[64];
        u_char  dl_module_id_2[64];
        u_char  dl_arpmod_name[64];
        u_char  dl_nmid;
        u_long  dl_reserved1;
        u_long  dl_reserved2;
    } dl_hp_ppa_info_t;
```

dl_next_offset

offset of next ppa info structure from start of info area.

dl_ppa

PPA # assigned to LAN interface.

dl_hw_path

hardware path of LAN interface.

dl_mac_type

MAC type of LAN interface.

dl_phys_addr

station address.

dl_addr_length

length of station address.

dl_mjr_num

major number of interface driver.

dl_name

name of driver.

dl_instance_num

instance number of device.

dl_mtu

MTU

dl_hdw_state

hardware state

dl_module_id_1

default module ID name for the stream. The default name is “lan.” This value is used as the interface name when executing the `ifconfig` command.

dl_module_id_2

optional module ID name for streams that support multiple encapsulation types. If the user is attached to a stream that supports ETHER and IEEE8023, then this name is set to “snap.” Otherwise, the field is set to NULL. This value is used as the interface name when executing the `ifconfig` command.

dl_arpmod_name

identifies the ARP helper module for the network interface. If the driver does not have an ARP helper, this field will be NULL.

dl_nmid

identifies the network management ID value for a specific interface.

dl_reserved[1,2]

reserved fields

State

The message is valid in any State in response to a DL_PPA_REQ.

New State

The resulting state is unchanged.

DL_INFO_REQ

Requests information of the DLS provider about the DLPI stream. This information includes a set of provider-specific parameters, as well as the current state of the interface.

The message consists of one M_PCPROTO message block, which contains the following structure.

Format

```
typedef struct {
        ulong        dl_primitive;
} dl_info_req_t;
```

Parameters

dl_primitive

DL_INFO_REQ

State

The message is valid in any state in which a local acknowledgment is not pending, as described in Appendix B, Allowable Sequence of DLPI Primitives, of the DLPI 2.0 specification.

New State

The resulting state is unchanged.

DL_INFO_ACK

This message is sent in response to DL_INFO_REQ; it conveys information about the DLPI stream to the DLS user.

This message consists of one M_PCPROTO message block, which contains the following structure.

Format

```
typedef struct {
       ulong       dl_primitve;
       ulong       dl_max_sdu;
       ulong       dl_min_sdu;
       ulong       dl_addr_length;
       ulong       dl_mac_type;
       ulong       dl_reserved;
       ulong       dl_current_state;
       ulong       dl_sap_length;
       ulong       dl_service_mode;
       ulong       dl_qos_length;
       ulong       dl_qos_offset;
       ulong       dl_qos_range_length;
       ulong       dl_provider_style;
       ulong       dl_addr_offset;
       ulong       dl_version;
       ulong       dl_brdcst_addr_length;
```

```
        ulong       dl_brdcst_addr_offset;
        ullong      dl_growth;
} dl_info_ack_t;
```

Parameters

dl_primitive

DL_INFO_ACK

dl_max_sdu

maximum number of bytes that may be transmitted in a data link service data unit (DLSDU). This value must be a positive integer that is greater than or equal to the value of dl_min_sdu.

dl_min_sdu

minimum number of bytes that may be transmitted in a DLSDU. The value is never less than one.

dl_addr_length

length, in bytes, of the provider's DLSAP address.

dl_mac_type

type of medium supported. Possible values:

DL_CSMACD

Carrier Sense Multiple Access with Collision Detection (ISO 8802/3).

DL_TPB

Token-Passing Bus (ISO 8802/4).

DL_TPR

Token-Passing Ring (ISO 8802/5).

DL_METRO

Metro Net (ISO 8802/6).

DL_ETHER

Ethernet Bus.

DL_HDLC

bit synchronous communication line.

DL_CHAR

character synchronous communication line.

DL_CTCA

channel-to-channel adapter.

DL_FDDI

Fiber Distributed Data Interface.

DL_OTHER

any other medium not listed above.

NOTE dl_mac_type is not valid until after a dl_attach_req has been issued.

dl_reserved

reserved field whose value must be set to zero.

dl_current_state

state of the DLPI interface for the stream when the DLS provider issued this acknowledgment.

dl_sap_length

current length of the SAP component of the DLSAP address. It may have a negative, zero or positive. A positive value indicates the ordering of the SAP and PHSYCAL component within the DLSAP address as SAP component followed by PHYSICAL component. A negative value indicates PHYSICAL followed by the SAP. A zero value indicates that no SAP has yet been bound. The absolute value of the dl_sap_length provides the length of the SAP component within the DLSAP address.

dl_service_mode

if returned before the DL_BIND_REQ is processed, this conveys which services modes the DLS provider can support. It contains a bit-mask specifying on or more than one of the following values:

DL_CODLS

connection-oriented data link service.

DL_CLDLS

connection-less data link service.

DL_HP_RAWDLS

raw-mode service.

DL_ACLDLS

acknowledged connectionless data link service.

Since ATM is a connection-oriented link, the value of this field will always be DL_CODLS.

dl_qos_length

length, in bytes, of the negotiated/selected values of the quality of service (QOS) parameters. The returned values are those agreed during the negotiation. If QOS has not yet been negotiated, default values will be returned; these values correspond to those that will be applied by the DLS provider on a connect request.

The QOS values are conveyed in the structures defined in the above sections in this chapter. For any parameter the DLS provider does not support or cannot determine, the corresponding entry will be set to DL_UNKNOWN.

dl_qos_offset

offset from the beginning of the M_PCPROTO block where the current QOS parameters begin.

dl_qos_range_length

length, in bytes, of the available range of QOS parameter values supported by the DLS provider. This the range available to the calling DLS user in a connect request. The range of available QOS values is conveyed in the structures defined in the following section in this chapter. For any parameter the DLS provider does not support or cannot determine, the corresponding entry will be set to DL_UNKNOWN.

dl_qos_range_offset

offset from the beginning of the M_PCPROTO block where the available range of quality of service parameters begins.

dl_provider_style

style of DLS provider associated with the DLPI stream. The following provider classes are defined.

DL_STYLE1

PPA is implicitly attached to the DLPI stream by opening the appropriate major/minor device number.

DL_STYLE2

DLS user must explicitly attach a PPA to the DLPI stream using DL_ATTACH_REQ.

ATM DLPI only supports DL_STYLE2.

dl_addr_offset

offset of the address that is bound to the associated stream. If the DLS user issues a DL_INFO_REQ prior to binding a DLSAP, the value of dl_addr_len will be 0 and consequently indicate that there has been no address bound.

dl_version

current supported version of the DLPI.

dl_brdcst_addr_length

length of the physical broadcast address. ATM DLPI does not support broadcast addresses and therefore, the value of this field will be zero.

dl_brdcst_addr_offset

not applicable to ATM DLPI.

dl_growth

growth field for future use. The value of this field will be zero.

State

The message is valid in any state in response to a DL_INFO_REQ.

New State

The resulting state is unchanged.

DL_ATTACH_REQ

Requests the DLS provider to associate a physical point of attachment (PPA) with a stream.

The message consists of one M_PROTO message block, which contains the following structure.

Format

```
typedef struct {
       ulong       dl_primitive;
       ulong       dl_ppa;
} dl_attach_req_t;
```

Parameters

dl_primitive

DL_ATTACH_REQ

dl_ppa

identifier of the physical point of attachment to be associated with the stream.

State

The message is valid in state DL_UNATTACHED.

New State

The resulting state is DL_ATTACH_PENDING.

Response

If the attach request is successful, DL_OK_ACK is sent to the DLS user resulting in state DL_UNBOUND.

If the request fails, DL_ERROR_ACK is returned and the resulting state is unchanged.

Reasons for Failure

DL_BADPPA

The specified PPA is invalid.

DL_ACCESS

The DLS user did not have proper permission to use the requested PPA.

DL_OUTSTATE

The primitive was issued from an invalid state.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_DETACH_REQ

Requests the DLS provider to disassociate a physical point of attachment (PPA) with a stream.

The message consists of one M_PROTO message block, which contains the following structure.

Format

```
typedef struct {
      ulong       dl_primitive;
} dl_detach_req_t;
```

Parameters

dl_primitive

DL_DETACH_REQ

State

The message is valid in state DL_UNBOUND.

New State

The resulting state is DL_DETACH_PENDING.

Response

If the detach request is successful, DL_OK_ACK is sent to the DLS user resulting in state DL_UNATTACHED.

If the request fails, DL_ERROR_ACK is returned and the resulting state is unchanged.

Reasons for Failure

DL_OUTSTATE

The primitive was issued from an invalid state.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_BIND_REQ

Requests the DLS provider to bind a DLSAP to the stream. The DLS user must identify the address of the DLSAP to be bound to the stream. The DLS user also indicates whether it will accept incoming connection requests on the stream. Finally, the request directs the DLS provider to activate the stream associated with the DLSAP.

The message consists of one M_PROTO message block, which contains the following structure.

Format

```
typedef struct {
       ulong          dl_primitive;
       ulong          dl_sap;
       ulong          dl_max_conind;
       ushort         dl_service_mode;
       ushort         dl_conn_mgmt;
       ulong          dl_xidtest_flg;
} dl_bind_req_t;
```

Parameters

dl_primitive

DL_BIND_REQ

dl_sap

DLSAP that will be bound to the DLPI stream.

dl_max_conind

maximum number of outstanding DL_CONNECT_IND messages allowed on the DLPI stream. If the value is zero, the stream cannot accept any DL_CONNECT_IND messages. If greater than zero, the DLS user will accept DL_CONNECT_IND messages up to the given value before having to respond with a DL_CONNECT_RES or a DL_DISCONNECT_REQ.

dl_service_mode

desired mode of service for this stream. This field should be set to one of the following:

DL_CODLS

connection-mode

DL_CLDLS

connectionless-mode

DL_HP_RAWDLS

raw-mode

dl_conn_mgmt

indicates that the stream is the “connection management” stream for the PPA to which the stream is attached. This field should be set to zero.

dl_xidtest_flg

indicates to the DLS provider that XID and/or TEST responses for this stream are to be automatically generated by the DLS Provider.

State

The message is valid in state DL_UNBOUND.

New State

The resulting state is DL_BIND_PENDING.

Response

If the bind request is successful, DL_BIND_ACK is sent to the DLS user resulting in state DL_IDLE.

If the request fails, DL_ERROR_ACK is returned and the resulting state is unchanged.

Reasons for Failure

DL_BADADDR

The DLSAP address information was invalid or was in an incorrect format.

DL_INITFAILED

Automatic initialization of the PPA failed.

DL_NOTINIT

The PPA had not been initialized prior to this request.

DL_ACCESS

The DLS user did not have proper permission to use the requested DLSAP address.

DL_BOUND

The DLS user attempted to bind a second stream to a DLSAP with dl_max_conind greater than zero, or the DLS user attempted to bind a second “connection management” stream to a PPA.

DL_OUTSTATE

The primitive was issued from an invalid state.

DL_NOADDR

The DLS provider could not allocate a DLSAP address for this stream.

DL_UNSUPPORTED

The DLS provider does not support requested service mode on this stream.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_NOAUTO

Automatic handling of XID and TEST responses not supported.

DL_NOXIDAUTO

Automatic handling of XID response not supported.

DL_OUTSTATE

The primitive was issued from an invalid state.

DL_BIND_ACK

Reports the successful bind of a DLSAP to a stream, and returns the bound DLSAP address to the DLS user. This primitive is generated in response of a DL_BIND_REQ.

The message consists of one M_PCPROTO message block, which contains the following structure.

Message Format

```
typedef struct {
       ulong          dl_primitive;
       ulong          dl_sap;
       ulong          dl_addr_length;
       ulong          dl_addr_offset;
       ulong          dl_max_conind;
       ulong          dl_xidtest_flg;
} dl_bind_ack_t;
```

Parameters

dl_primitive

DL_BIND_ACK

dl_sap

DLSAP address information associated with the bound DLSAP. It corresponds to the dl_sap field of the associated DL_BIND_REQ, which contains part of the DLSAP address.

dl_addr_length

length of the complete DLSAP address that was bound to the DLPI stream.

dl_addr_offset

offset from the beginning of the M_PCPROTO block where the DLSAP address begins.

dl_max_conind

allowed maximum number of outstanding DL_CONNECT_IND messages to be supported on the DLPI stream.

dl_xidtest_flg

XID and TEST responses supported by the provider.

State

The message is valid in state DL_BIND_PENDING.

New State

The resulting state is DL_IDLE.

DL_UNBIND_REQ

Requests the DLS provider to unbind the DLSAP that had been bound by a previous DL_BIND_REQ from this stream.

The message consists of one M_PROTO message block, which contains the following structure.

Format

```
typedef struct {
      ulong          dl_primitive;
} dl_unbind_req_t;
```

Parameters

dl_primitive

DL_UNBIND_REQ

State

The message is valid in state DL_IDLE.

New State

The resulting state is DL_UNBIND_PENDING.

Response

If the unbind request is successful, DL_OK_ACK is sent to the DLS user resulting in state DL_UNBOUND.

If the request fails, DL_ERROR_ACK is returned and the resulting state is unchanged.

Reasons for Failure

DL_OUTSTATE

The primitive was issued from an invalid state.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_SUBS_BIND_REQ

Requests the DLS provider bind a subsequent DLSAP to the stream. The DLS user must identify the address of the subsequent DLSAP to be bound to the stream.

Format

The message consists of one M_PROTO message block, which contains the following structure.

```
typedef struct {
     ulong        dl_primitive;
     ulong        dl_subs_sap_offset;
     ulong        dl_subs_sap_length;
     ulong        dl_subs_bind_class;
} dl_subs_bind_req_t;
```

Parameters

dl_primitive

DL_SUBS_BIND_REQ

dl_subs_sap_offset

offset of the DLSAP from the beginning of the M_PROTO block.

dl_subs_sap_length

length of the specified DLSAP.

dl_subs_bind_class

specifies either peer or hierarchical addressing.

DL_PEER_BIND

specifies peer addressing. The DLSAP specified is used in lieu of the DLSAP bound in the BIND request.

DL_HIERARCHICAL_BIND

specifies hierarchical addressing. The DLSAP specified is used in addition to the DLSAP specified using the BIND request.

State

The message is valid in state DL_IDLE.

New State

The resulting state is DL_SUBS_BIND_PND.

Response

If the subsequent bind request is successful, DL_SUBS_BIND_ACK is sent to the DLS user resulting in state DL_IDLE.

Reasons for Failure

DL_BADADDR

The DLSAP address information was invalid or was in an incorrect format.

DL_ACCESS

The DLSAP user did not have proper permission to use the requested DLSAP address.

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_UNSUPPORTED

Requested addressing class not supported.

DL_TOOMANY

Limit exceeded on the maximum number of DLSAPs per stream.

DL_SUBS_BIND_ACK

Reports the successful bind of a subsequent DLSAP to a stream, and returns the bound DLSAP address to the DLS user. This primitive is generated in response to a DL_SUBS_BIND_REQ.

Format

The message consists of one M_PROTO message block, which contains the following structure.

```
typedef struct  {
     ulong           dl_primitive;
     ulong           dl_subs_sap_offset;
     ulong           dl_subs_sap_length;
}  dl_subs_bind_ack_t;
```

Parameters

dl_primitive

DL_SUBS_BIND_ACK

dl_subs_sap_offset

offset of the DLSAP from the beginning of the M_PCPROTO block.

dl_subs_sap_length

length of the specified DLSAP.

State

The message is valid in state DL_SUBS_BIND_PND.

New State

The resulting state is DL_IDLE.

DL_SUBS_UNBIND_REQ

Requests the DLS provider to unbind the DLSAP that had been bound by a previous DL_SUBS_BIND_REQ from this stream.

Format

The message consists of one M_PROTO message block, which contains the following structure.

```
typedef struct  {
     ulong           dl_primitive;
     ulong           dl_subs_sap_offset;
     ulong           dl_subs_sap_length;
}  dl_subs_unbind_req_t;
```

Parameters

dl_primitive

DL_SUBS_UNBIND_REQ

dl_subs_sap_offset

offset of the DLSAP from the beginning of the M_PROTO block.

dl_subs_sap_length

length of the specified DLSAP.

State

The message is valid in state DL_IDLE.

New State

The resulting state is DL_SUBS_UNBIND_PND.

Response

If the unbind request is successful, a DL_OK_ACK is sent to the DLS User. The resulting state is DL_IDLE.

If the request fails, DL_ERROR_ACK is returned and the resulting state is unchanged.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_BADADDR

The DLSAP address information was invalid or was in an incorrect format.

DL_ENABMULTI_REQ

Requests the DLS Provider to enable specific multicast addresses on a per Stream basis. It is invalid for a DLS Provider to pass upstream messages that are destined for any address other than those explicitly enabled on that Stream by the DLS User.

Format

The message consists of one M_PROTO message block, which contains the following structure:

```
typedef struct  {
     ulong           dl_primitive;
     ulong           dl_addr_length;
     ulong           dl_addr_offset;
} dl_enabmulti_req_t;
```

Parameters

dl_primitive

DL_ENABMULTI_REQ

dl_addr_length

length of the multicast address.

dl_addr_offset

offset from the beginning of the M_PROTO message block where the multicast address begins.

State

This message is valid in any state in which a local acknowledgment is not pending with the exception of DL_UNATTACH.

New State

The resulting state is unchanged.

Response

If the enable request is successful, a DL_OK_ACK is sent to the DLS user. If the request fails, DL_ERROR_ACK is returned and the resulting state is unchanged.

Reasons for Failure

DL_BADADDR

Address information was invalid or was in an incorrect format.

DL_TOOMANY

Too many multicast address enable attempts. Limit exceeded.

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_NOTSUPPORTED

Primitive is known, but not supported by the DLS Provider.

DL_DISABMULTI_REQ

Requests the DLS Provider to disable specific multicast addresses on a per Stream basis.

Format

The message consists of one M_PROTO message block, which contains the following structure:

```
typedef struct  {
     ulong           dl_primitive;
     ulong           dl_addr_length;
     ulong           dl_addr_offset;
}  dl_disabmulti_req_t;
```

Parameters

dl_primitive

DL_DISABMULTI_REQ

dl_addr_length

length of the physical address.

dl_addr_offset

offset form the beginning of the M_PROTO message block where the multicast address begins.

State

This message is valid in any state in which a local acknowledgment is not pending with the exception of DL_UNATTACH.

New State

The resulting state is unchanged.

Response

If the disable request is successful, a DL_OK_ACK is sent to the DLS user. If the request fails, DL_ERROR_ACK is returned and the resulting state is unchanged.

Reasons for Failure

DL_BADADDR

Address information was invalid or in an incorrect format.

DL_NOTENAB

Address specified is not enabled.

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_NOTSUPPORTED

Primitive is known, but not supported by the DLS Provider.

DL_PROMISCON_REQ

This primitive requests the DLS Provider to enable promiscuous mode on a per Stream basis, either at the physical level or at the SAP level.

The DL Provider will route all received messages on the media to the DLS User until either a DL_DETACH_REQ or a DL_PROMISCOFF_REQ is received or the Stream is closed.

Format

The message consists of one M_PROTO message block, which contains the following structure.

```
typedef struct  {
     ulong           dl_primitive;
     ulong           dl_level;
}  dl_promiscon_req_t;
```

Parameters

dl_primitive

DL_PROMISCON_REQ

dl_level

indicates promiscuous mode at the physical or SAP level.

DL_PROMISC_PHYS

Before or after the STREAM has been bound, the DLPI user receives all traffic on the wire regardless of protocol or physical address.

DL_PROMISC_SAP

Before or after the STREAM has been bound, the DLPI user receives all traffic destined for this interface (physical addresses, broadcast addresses or bound multicast addresses) that matches any protocol enabled on that interface.

DL_PROMISC_MULTI

Before or after the STREAM has been bound, the DLPI user receives all multicast packets on the wire regardless of the protocol it is destined for.

State

The message is valid in any state when there is no pending acknowledgment.

New State

The resulting state is unchanged.

Response

If enabling of promiscuous mode is successful, a DL_OK_ACK is returned. Otherwise, a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_NOTSUPPORTED

Primitive is known but not supported by the DLS Provider.

DL_UNSUPPORTED

Requested service is not supplied by the provider.

DL_PROMISCOFF_REQ

This primitive requests the DLS Provider to disable promiscuous mode on a per Stream basis, either at the physical level or at the SAP level.

Format

The message consists of one M_PROTO message block, which contains the following structure.

```
typedef struct  {
     ulong          dl_primitive;
     ulong          dl_level;
}  dl_promiscoff_req_t;
```

Parameters

dl_primitive

DL_PROMISCOFF_REQ

dl_level

indicates promiscuous mode at the physical or SAP level.

DL_PROMISC_PHYS

Before or after the STREAM has been bound, the DLPI user receives all traffic on the wire regardless of protocol or physical address.

DL_PROMISC_SAP

Before or after the STREAM has been bound, the DLPI user receives all traffic destined for this interface (physical addresses, broadcast addresses or bound multicast addresses) that matches any protocol enabled on that interface.

DL_PROMISC_MULTI

Before or after the STREAM has been bound, the DLPI user receives all multicast packets on the wire regardless of the protocol it is destined for.

State

The message is valid in any state in which the promiscuous mode is enabled and there is no pending acknowledgment.

New State

The resulting state is unchanged.

Response

If the promiscuous mode disabling is successful, a DL_OK_ACK is returned. Otherwise, a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_NOTSUPPORTED

Primitive is known but not supported by the DLS Provider.

DL_NOTENAB

Mode not enabled.

DL_OK_ACK

Acknowledges to the DLS user that a previously issued request primitive was received successfully. It is only initiated for those primitives that require a positive acknowledgment.

Format

The message consists of one M_PCPROTO message block, which contains the following structure.

```
typedef struct {
      ulong          dl_primitive;
      ulong          dl_correct_primitve;
} dl_ok_ack_t;
```

Parameters

dl_primitive

DL_OK_ACK

dl_correct_primitive

identifies the successfully received primitive that is being acknowledged.

State

The message is valid in response to a DL_ATTACH_REQ, DL_DETACH_REQ, DL_UNBIND_REQ, DL_CONNECT_RES, DL_RESET_RES, DL_DISCON_REQ, DL_SUBS_UNBIND_REQ, DL_PROMISCON_REQ, DL_ENABMULTI_REQ, DL_DISADMULTI_REQ or DL_PROMISCOFF_REQ from any of several states as defined in Appendix B, Allowable Sequence of DLPI Primitives, of the DLPI 2.0 specification.

New State

The resulting state depends on the current state and is defined fully in Appendix B, Allowable Sequence of DLPI Primitives, of the DLPI 2.0 specification.

DL_ERROR_ACK

Informs the DLS user that the previous request or response was invalid.

Format

The message consists of one M_PCPROTO message block, which contains the following structure.

```
typedef struct {
      ulong          dl_primitive;
      ulong          dl_error_primitive;
```

```
      ulong          dl_errno;
      ulong          dl_unix_errno;
} dl_error_ack_t;_
```

Parameters

dl_primitive

DL_ERROR_ACK

dl_error_primitive

primitive that is in error.

dl_errno

DLPI error code associated with the failure.

dl_unix_errno

UNIX system error code associated with the failure. This value should be non-zero only when dl_errno is set to DL_SYSERR. It is used to report UNIX system failures that prevent the processing of a given request or response.

State

The message is valid in every state where an acknowledgement or confirmation of a previous request or response is pending.

New State

The resulting state is that from which the acknowledged request or response was generated.

Optional Primitives to Perform Essential Management Functions

This section describes optional primitives. Some of these primitives may not be supported by the DLS provider.

DL_PHYS_ADDR_REQ

Requests the DLS provider to return the physical address associated with the stream depending upon the value of the address type selected in the request.

Format

The message consists one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong          dl_primitive;
      ulong          dl_addr_type;
} dl_phys_addr_req_t;
```

Parameters

dl_primitive

DL_PHYS_ADDR_REQ

dl_addr_type

type of address requested - factory physical address or current physical address

DL_FACT_PHYS_ADDR

DL_CURR_PHYS_ADDR

State

The message is valid in any attached state in which a local acknowledgement is not pending. For a style 2 provider, this would be after a PPA is attached using the DL_ATTACH_REQ. For a style 1 provider, the PPA is implicitly attached after the stream is opened.

New State

The resulting state is unchanged.

Response

The provider responds to the request with a DL_PHYS_ADDR_ACK if the request is supported. Otherwise, a DL_ERROR_ACK is returned.

Reasons for Failure

DL_NOTSUPPORTED

The primitive is known, but not supported by the DLS provider.

DL_OUTSTATE

The primitive was issued from an invalid state.

DL_PHYS_ADDR_ACK

This primitive returns the value for the physical address to the link user in response to a DL_PHYS_ADDR_REQ.

Format

The message consists one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong           dl_primitive;
      ulong           dl_addr_length;
      ulong           dl_addr_offset;
} dl_phys_addr_ack_t;
```

Parameters

dl_primitive

DL_PHYS_ADDR_ACK

dl_addr_length

length of the requested hardware address.

dl_addr_offset

offset from beginning of the M_PROTO message block.

State

The message is valid in any state in response to a DL_PHYS_ADDR_REQ.

New State

The resulting state is unchanged.

DL_SET_PHYS_ADDR_REQ

Sets the physical address value for all streams for that provider for a particular PPA.

Format

The message consists one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong           dl_primitive;
      ulong           dl_addr_length;
      ulong           dl_addr_offset;
} dl_set_phys_addr_req_t;
```

Parameters

dl_primitive

DL_SET_PHYS_ADDR_REQ

dl_addr_length

length of the requested hardware address.

dl_addr_offset

offset from beginning of the M_PROTO message block.

State

The message is valid in any attached state in which a local acknowledgement is not pending. For a style 2 provider, this would be after a PPA is attached using the DL_ATTACH_REQ. For a style 1 provider, the PPA is implicitly attached after the stream is opened.

New State

The resulting state is unchanged.

Response

The provider responds to the request with a DL_OK_ACK on successful completion. Otherwise, a DL_ERROR_ACK is returned.

Reasons for Failure

DL_BADADDR

The address information was invalid or was in an incorrect format.

DL_NOTSUPPORTED

The primitive is known, but not supported by the DLS provider.

DL_SYSERR

A system error has occurred.

DL_OUTSTATE

The primitive was issued from an invalid state.

DL_BUSY

One or more streams for that particular PPA are in the DL_BOUND state.

DL_GET_STATISTICS_REQ

Directs the DLS provider to return statistics.

Format

The message consists one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong        dl_primitive;
} dl_get_statistics_req_t;
```

Parameters

dl_primitive

DL_GET_STATISTICS_REQ

State

The message is valid in any attached state in which a local acknowledgement is not pending.

New State

The resulting state is unchanged.

Response

The DLS provider responds to the request with a DL_GET_STATISTICS_ACK if the primitive is supported. Otherwise, a DL_ERROR_ACK is returned.

Reasons for Failure

DL_NOTSUPPORTED

The primitive is known, but not supported by the DLS provider.

DL_GET_STATISTICS_ACK

Returns statistics in response to the DL_GET_STATISTICS_REQ. The content of this statistics block is the following:

Format

The message consists one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong       dl_primitive;
      ulong       dl_stat_length;
      ulong       dl_stat_offset;
} dl_get_statistics_ack_t;
```

Parameters

dl_primitive

DL_GET_STATISTICS_ACK

dl_stat_length

length of the statistics structure.

dl_stat_offset

offset from the beginning of the M_PCPROTO message block where the statistics information resides.

State

The message is valid in any state in response to a DL_GET_STATISTICS_REQ.

New State

The resulting state is unchanged.

The DL_GET_STATISTICS_ACK returns standard mib and optionally extended mib information for all HP supported networking interfaces. It is up to the DLPI user to check the interface-specific field of the Interface MIB to determine whether there is a transmission MIB.

DL_HP_MULTICAST_LIST_REQ

Requests the DLS Provider to return a list of all currently enabled multicast addresses on a specific LAN interface.

Format

The message consists one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong       dl_primitive;
} dl_hp_multicast_list_req_t;_
```

Parameters

dl_primitive

DL_HP_MULTICAST_LIST_REQ

State

The message is valid in any state in which there is not a local acknowledgment pending with the exception of DL_UNATTACH.

New State

The resulting state is unchanged.

Response

If the multicast request is successful, a DL_HP_MULTICAST_LIST_ACK is sent to the DLS user. If the requests fails, DL_ERROR_ACK is returned and the resulting state is unchanged.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_HP_MULTICAST_LIST_ACK

Reports the successful completion of a DL_HP_MULTICAST_LIST_REQ primitive. A complete list of the multicast addresses for a specific LAN interface are returned after the control message header.

Format

The message consists one M_PROTO message block containing the structure shown below.

```
typedef struct {
       ulong     dl_primitive;
       ulong     dl_offset;
       ulong     dl_length;
       ulong     dl_count;
} dl_hp_multicast_list_ack_t;
```

Parameters

dl_primitive

DL_HP_MULTICAST_LIST_ACK

dl_offset

offset to the data in the multicast acknowledgment.

dl_length

length of data area, in bytes.

dl_count

total number of 6 byte multicast addresses in the data area of the multicast acknowledgment.

State

The message is valid in any state in response to a DL_HP_MULTICAST_LIST_REQ.

New State

The resulting state is unchanged.

Connectionless-mode Service Primitives

This section describes the connectionless-mode service primitives.

DL_UNITDATA_REQ

Conveys one DLSDU from the DLS user to the DLS provider for transmission to a peer DLS user.

Because connectionless data transfer is an unacknowledged service, the DLS provider makes no guarantees of delivery of connectionless DLSDUs. It is the responsibility of the DLS user to do any necessary sequencing or retransmission of DLSDUs in the event of a presumed loss.

Priority messages are currently only supported over 100VG. To send a priority message over 100VG, a user must have superuser capabilities and set the dl_priority fields in the DL_UNITDATA_REQ primitive to the following values:

dl_min must be set to 0.

dl_max must be set to 1.

The dl_priority field will be ignored on interfaces which do not support priority messages.

Format

The message consists of one M_PROTO message block containing the structure shown below, followed by one or more M_DATA blocks containing at least one byte of data. The amount of user data that may be transferred in a single DLSDU is limited. This limit is conveyed by the parameter dl_max_sdu in the DL_INFO_ACK primitive.

```
typedef  struct  {
      ulong          dl_primitive;
      ulong          dl_dest_addr_length;
      ulong          dl_dest_addr_offset;
      dl_priority_t  dl_priority;
}  dl_unitdata_req_t;
```

Parameters

dl_primitive

DL_UNITDATA_REQ

dl_dest_addr_length

length of the DLSAP address of the destination DLS user. If the destination user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

dl_priority

priority value within the supported range for this particular DLSDU.

State

The message is valid in state DL_IDLE.

New State

The resulting state is unchanged.

Response

If the DLS provider accepts the data for transmission, there is no response. This does not, however, guarantee that the data will be delivered to the destination DLS user, since the connectionless data transfer is not a confirmed service.

If the request is erroneous, DL_UDERROR_IND is returned, and the resulting state is unchanged.

If for some reason the request cannot be processed, the DLS provider may generate a DL_UDERROR_IND to report the problem. There is, however, no guarantee that such an error report will be generated for all undeliverable data units, since connectionless data transfer is not a confirmed service.

Reasons for Failure

DL_BADADDR

The destination DLSAP address was in an incorrect format or contained invalid information.

DL_BADDATA

The amount of data in the current DLSDU exceeded the DLS provider's DLSDU limit.

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_UNSUPPORTED

Requested priority not supplied by provider.

DL_UNITDATA_IND

Conveys one DLSDU from the DLS provider to the DLS user.

Format

The message consists of one M_PROTO message block containing the structure shown below, followed by one or more M_DATA blocks containing at least one byte of data. The amount of user data that may be transferred in a single DLSDU is limited. This limit is conveyed by the parameter dl_max_sdu in the DL_INFO_ACK primitive.

```
typedef  struct  {
      ulong          dl_primitive;
      ulong          dl_dest_addr_length;
      ulong          dl_dest_addr_offset;
      ulong          dl_src_addr_length;
      ulong          dl_src_addr_offset;
      ulong          dl_group_address;
 }  dl_unitdata_ind_t;
```

Parameters

dl_primitive

DL_UNITDATA_IND

dl_dest_addr_length

length of the address of the DLSAP where this DL_UNITDATA_IND is intended to be delivered.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

dl_src_addr_length

length of the DLSAP address of the sending DLS user.

dl_src_addr_offset

offset from the beginning of the M_PROTO message block where the source DLSAP address begins.

dl_group_address

is set by the DLS provider upon receiving and passing upstream a data message when the destination address of the data message is a multicast or broadcast address.

State

The message is valid in any attached state.

New State

The resulting state is unchanged.

DL_UDERROR_IND

Informs the DLS user that a previously sent DL_UNITDATA_REQ produced an error or could not be delivered. The primitive indicates the destination DLSAP address associated with the failed request, and conveys an error value that specifies the reason for failure.

Format

The message consists of either one M_PROTO message block or one M_PCPROTO message block containing the structure shown below.

```
typedef  struct  {
      ulong          dl_primitive;
      ulong          dl_dest_addr_length;
      ulong          dl_dest_addr_offset;
      ulong          dl_unix_errno;
      ulong          dl_errno;
} dl_uderror_ind_t;
```

Parameters

dl_primitive

DL_UDERROR_IND

dl_dest_addr_length

length of the DLSAP address of the destination DLS user.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

dl_unix_errno

UNIX system error code associated with the failure. This value should be non-zero only when dl_errno is set to DL_SYSERR. It is used to report UNIX system failures that prevent the processing of a given request.

dl_errno

DLPI error code associated with the failure. See Reasons for Failure in the description of DL_UNITDATA_REQ for the error codes that apply to an erroneous DL_UNITDATA_REQ. In addition, the error value DL_UNDELIVERABLE may be returned if the request was valid but for some reason the DLS provider could not deliver the data unit (e.g. due to lack of sufficient local buffering to store the data unit). There is, however, no guarantee that such an error report will be generated for all undeliverable data units, since connectionless data transfer is not a confirmed service.

State

The message is valid in state DL_IDLE.

New State

The resulting state is unchanged.

Raw Mode Service Primitives

This section describes the raw mode service primitives.

DL_HP_RAWDATA_REQ

Requests the DLS provider to send one completely formatted DLSDU to a peer DLS user. The DLSDU is assumed to have a complete Link and MAC Level header included.

As with connectionless data transfer, raw mode is an unacknowledged service, and the DLS provider makes no guarantees of delivery of connectionless DLSDUs. It is the responsibility of the DLS user to do any necessary sequencing or retransmission of DLSDUs in the event of a presumed loss.

Format

The message consists of one M_PROTO message block containing the structure shown below, followed by one or more M_DATA message blocks containing at least one byte of data. The amount of user data that may be transferred in a single DLSDU is limited. This limit is conveyed by the parameter dl_max_sdu in the DL_INFO_ACK primitive.

```
typedef  struct  {
     ulong          dl_primitive;
} dl_hp_rawdata_req_t;
```

Parameters

dl_primitive

DL_HP_RAWDATA_REQ

State

The message is valid in state DL_IDLE.

New State

The resulting state is unchanged.

Response

If the DLS provider accepts the data for transmission, there is no response. This does not, however, guarantee that the data will be delivered to the destination DLS user, since the connectionless data transfer is not a confirmed service.

If the request is erroneous, a DL_ERROR_ACK is returned, and the resulting state is unchanged.

Reasons for Failure

DL_BADPRIM

Request was issued from a state in which the DL_HP_RAWDATA_REQ was not recognized.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_HP_RAWDATA_IND

Conveys one completely formatted DLSDU from the DLS provider to the DLS user. The DLSDU contains the complete Link and MAC Level headers.

Format

The message consists of one M_PROTO message block containing the structure shown below, followed by one or more M_DATA message blocks containing at least one byte of data. The amount of user data that may be transferred in a single DLSDU is limited. This limit is conveyed by the parameter dl_max_sdu in the DL_INFO_ACK primitive.

```
typedef  struct  {
     ulong          dl_primitive;
} dl_hp_rawdata_ind_t;
```

Parameters

dl_primitive

DL_HP_RAWDATA_IND

State

The message is valid in state DL_IDLE.

New State

The resulting state is unchanged.

Connection-mode Service Primitives

This section describes the service primitives that support the connection-mode service of the data link layer. These primitives support the establishment of connections, connection-mode data transfer, and connection release services.

In the connection establishment model, the calling DLS user initiates a request for a connection, and the called DLS user receives each request and either accepts or rejects it. In the simplest form, the called DLS user is passed a connect indication and the DLS provider holds any subsequent indications until a response for the current outstanding indication is received. At most one connect indication is outstanding at any time.

DLPI also enables a called DLS user to multi-thread connect indications and responses. The DLS provider will pass all connect indications to the called DLS user (up to some pre-established limit as set by DL_BIND_REQ and DL_BIND_ACK). The called DLS user may then respond to the requests in any order.

To support multi-threading, a correlation value is needed to associate responses with the appropriate connect indication. A correlation value is contained in each DL_CONNECT_IND, and the DLS user must use this value in the DL_CONNECT_RES or DL_DISCONNECT_REQ primitive used to accept or reject the connect request.

Once a connection has been accepted or rejected, the correlation value has no meaning to a DLS user. The DLS provider may reuse the correlation value in another DL_CONNECT_IND.

Connection-Oriented DLPI Extensions

These primitives are only valid on connection-oriented DLPI STREAMS. Connection-oriented DLPI streams are those on which a DL_BIND_REQ with dl_service_mode set to DL_CODLS has been done.

DL_HP_INFO_REQ

Requests the DLS provider to provide information on the state of the connection on a DLPI stream.

Format

```
typedef struct {
     u_long        dl_primitive;
} dl_hp_info_req_t;
```

Parameters

dl_primitive

DL_HP_INFO_REQ

State

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and DL_PROV_RESET_PENDING.

New State

The resulting state is unchanged.

Response

If the primitive is issued from a valid state, the DLS provider responds with a DL_HP_INFO_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_HP_INFO_ACK

This message is sent in response to a DL_HP_INFO_REQ; it conveys information on the state of the connection on a DLPI stream.

Format

```
typedef struct {
    u_long  dl_primitive;
    u_long  dl_mem_fails;
    u_long  dl_queue_fails;
    u_long  dl_ack_to;
    u_long  dl_p_to;
    u_long  dl_rej_to;
    u_long  dl_busy_to;
    u_long  dl_send_ack_to;
    u_long  dl_ack_to_cnt;
    u_long  dl_p_to_cnt;
    u_long  dl_rej_to_cnt;
    u_long  dl_busy_to_cnt;
    u_long  dl_local_win;
```

```
    u_long  dl_remote_win;
    u_long  dl_i_pkts_in;
    u_long  dl_i_pkts_in_oos;
    u_long  dl_i_pkts_in_drop;
    u_long  dl_i_pkts_out;
    u_long  dl_i_pkts_retrans;
    u_long  dl_s_pkts_in;
    u_long  dl_s_pkts_out;
    u_long  dl_u_pkts_in;
    u_long  dl_u_pkts_out;
    u_long  dl_bad_pkts;
    u_long  dl_retry_cnt;
    u_long  dl_max_retry_cnt;
    u_long  dl_max_retries;
    u_long  dl_ack_thresh;
    u_long  dl_remote_busy_cnt;
    u_long  dl_hw_req_fails;
} dl_hp_info_ack_t;
```

Parameters

dl_primitive

DL_HP_INFO_ACK

dl_mem_fails

number of memory allocations that have failed.

dl_queue_fails

number of times that the DLS provider was unable to forward a message because the queue was full.

dl_ack_to

length of the ACK timeout in tenths of a second. The ACK timeout determines the length of time that LLC Type 2 will wait for an acknowledgment of any outstanding I PDUs or for a response to a U PDU before attempting to force a response.

dl_p_to

length of the P timeout in tenths of a second. The P timeout determines the length of time that LLC Type 2, after sending a command with the P bit set to 1, will wait for a response with the F bit set to 1 before attempting to force a response.

dl_rej_to

length of the REJ timeout in tenths of a second. The REJ timeout determines the length of time that LLC Type 2 will wait for a response to a REJ PDU before attempting to force a response.

dl_busy_to

length of the BUSY timeout in tenths of a second. The BUSY timeout determines the length of time that LLC Type 2 will wait for an indication that a remote busy condition has been cleared before attempting to force a response.

dl_send_ack_timeout

length of the SEND_ACK timeout in tenths of a second. The SEND_ACK timeout determines the maximum length of time that LLC Type 2 will delay acknowledgment of I PDUs if it has not received dl_send_ack_threshold I PDUs.

dl_ack_to_cnt

number of times that the ACK timer has expired.

dl_p_to_cnt

number of times that the P timer has expired.

dl_rej_to_cnt

number of times that the REJ timer has expired.

dl_busy_to_cnt

number of times that the BUSY timer has expired.

dl_local_win

size of the LLC Type 2 local receive window.

dl_remote_win

size of the LLC Type 2 remote receive window.

dl_i_pkts_in

number of I PDUs correctly received.

dl_i_pkts_in_oos

number of I PDUs received out of sequence.

dl_i_pkts_in_drop

number of I PDUs correctly received, but which were dropped because of a lack of resources.

dl_i_pkts_out

number of I PDUs acknowledged by the remote system.

dl_i_pkts_retrans

number of I PDUs re-transmitted.

dl_s_pkts_in

number of S PDUs received.

dl_s_pkts_out

number of S PDUs transmitted.

dl_u_pkts_in

number of U PDUs received.

dl_u_pkts_out

number of U PDUs transmitted.

dl_bad_pkts

number of PDUs with bad control fields received.

dl_retry_cnt

most recent number of times that LLC Type 2 has attempted to force a response from the remote due to a timer expiration. This value is re-set to 0 when a response is received.

dl_max_retry_cnt

maximum value that dl_retry_cnt has attained.

dl_max_retries

maximum allowed number of retries before re-setting the connection. This is sometimes known as the N2 variable.

dl_ack_thresh

maximum number of I PDUs that can be received before an acknowledgment is sent. If this threshold is reached, an acknowledgment is sent and the SEND_ACK timer is restarted.

dl_remote_busy_cnt

number of times that the remote system has reported that it was busy.

dl_hw_req_fails

number of times that LLC Type 2 has been unable to transmit due to congestion in the interface device driver or interface card.

State

The message is valid in any state in response to a DL_HP_INFO_REQ.

New State

The resulting state is unchanged.

DL_HP_SET_ACK_TO_REQ

Requests the DLS provider to set the ACK timeout to the specified value.

Format

```
typedef struct {
    u_long      dl_primitive;
    u_long      dl_ack_to;
} dl_hp_set_ack_to_req_t;
```

Parameters

dl_primitive

DL_HP_SET_ACK_TO_REQ

dl_ack_to

new value of the ACK timeout in tenths of a second. The ACK timeout determines the length of time that LLC Type 2 will wait for an acknowledgment of any outstanding I PDUs or for a response to a U PDU before attempting to force a response.

State

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and Dl_PROV_RESET_PENDING.

New State

The resulting state is unchanged.

Response

If the primitive is issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_HP_SET_P_TO_REQ

Requests the DLS provider to set the P timeout to the specified value.

Format

```
typedef struct {
    u_long  dl_primitive;
    u_long  dl_p_to;
} dl_hp_set_p_to_req_t;
```

dl_primitive

DL_HP_SET_P_TO_REQ

dl_p_to

new value of the P timeout in tenths of a second. The P timeout determines the length of time that LLC Type 2, after sending a command with the P bit set to 1, will wait for a response with the F bit set to 1 before attempting to force a response.

State

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and DL_PROV_RESET_PENDING.

New State

The resulting state is unchanged.

Response

If the primitive is issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_HP_SET_REJ_TO_REQ

Requests the DLS provider to set the REJ timeout to the specified value.

Format

```
typedef struct {
    u_long  dl_primitive;
    u_long  dl_rej_to;
} dl_hp_set_rej_to_req_t;
```

Parameters

dl_primitive

DL_HP_SET_REJ_TO_REQ

dl_rej_to

new value of the REJ timeout in tenths of a second. The REJ timeout determines the length of time that LLC Type 2 will wait for a response to a REJ PDU before attempting to force a response.

State

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and DL_PROV_RESET_PENDING.

New State

The resulting state is unchanged.

Response

If the primitive is issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_HP_SET_BUSY_TO_REQ

Requests the DLS provider to set the BUSY timeout to the specified value.

Format

```
typedef struct {
    u_long  dl_primitive;
    u_long  dl_busy_to;
} dl_hp_set_busy_to_req_t;
```

Parameters

dl_primitive

DL_HP_SET_BUSY_TO_REQ

dl_busy_to

new value of the BUSY timeout in tenths of a second. The BUSY timeout determines the length of time that LLC Type 2 will wait for an indication that a remote busy condition has been cleared before attempting to force a response.

State

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and DL_PROV_RESET_PENDING.

New State

The resulting state is unchanged.

Response

If the primitive is issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_HP_SET_SEND_ACK_TO_REQ

Requests the DLS provider to set the SEND_ACK timeout to the specified value.

Format

```
typedef struct {
    u_long  dl_primitive;
    u_long  dl_send_ack_to;
} dl_hp_set_send_ack_to_req_t;
```

Parameters

dl_primitive

DL_HP_SET_SEND_ACK_TO_REQ

dl_send_ack_to

new value of the SEND_ACK timeout in tenths of a second. The SEND_ACK timeout determines the maximum length of time that LLC Type 2 will delay acknowledgment of I PDUs if it has not received dl_send_ack_threshold I PDUs.

State

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and DL_PROV_RESET_PENDING.

New State

The resulting state is unchanged.

Response

If the primitive is issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_HP_SET_MAX_RETRIES_REQ

Requests the DLS provider to set the maximum allowed number of retries to the specified value.

Format

```
typedef struct {
    u_long      dl_primitive;
    u_long      dl_max_retries;
} dl_hp_set_max_retries_req_t;
```

Parameters

dl_primitive

DL_HP_SET_MAX_RETRIES_REQ

dl_max_retries

maximum allowed number of retries before re-setting the connection. This is sometimes known as the N2 variable.

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and DL_PROV_RESET_PENDING.

New State

The resulting state is unchanged.

Response

If the primitive is issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_HP_SET_ACK_THRESH_REQ

Requests the DLS provider to set the acknowledgment threshold to the specified value.

NOTE Setting the ack thresh will not affect the local window size.

Format

```
typedef struct {
    u_long      dl_primitive;
    u_long      dl_ack_thresh;
} dl_hp_set_ack_thresh_req_t;
```

Parameters

dl_primitive

DL_HP_SET_ACK_THRESH_REQ

dl_ack_thresh

maximum number of I PDUs that can be received before an acknowledgment is sent. If this threshold is reached, an acknowledgment is sent and the SEND_ACK timer is restarted. This value cannot be greater than the remote receive window size.

State

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and DL_PROV_RESET_PENDING.

New State

The resulting state is unchanged.

Response

If the specified dl_ack_thresh is valid and the primitive was issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_SYSERR

If the specified dl_ack_thresh is greater than the remote receive window size, then a DL_ERROR_ACK with dl_errno set to DL_SYSERR and dl_unix_errno set to EINVAL is returned.

DL_HP_SET_LOCAL_WIN_REQ

Requests the DLS provider to set the local window size to the specified value.

NOTE Setting the local window size also causes the DLPI read side streams queue hi water mark to be set to (local_window_size * MTU). The (local_window_size * MTU) cannot exceed (1 << 16) - (2 * MTU).

Format

```
typedef struct {
    u_long      dl_primitive;
    u_long      dl_local_win;
} dl_hp_set_local_win_req_t;
```

Parameters

dl_primitive

DL_HP_SET_LOCAL_WIN_REQ

dl_local_win

size of the local receive window. This value must be greater than 0 and less than 128.

State

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and DL_PROV_RESET_PENDING.

New State

The resulting state is unchanged.

Response

If the specified dl_local_win is valid and the primitive was issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_SYSERR

If the specified dl_local_win is invalid, then a DL_ERROR_ACK with dl_errno set to DL_SYSERR and dl_unix_errno set to EINVAL is returned.

DL_HP_SET_REMOTE_WIN_REQ

Requests the DLS provider to set the remote window size to the specified value.

NOTE Setting the remote window size causes the ack thresh to be set to ((remote_window_size + 1) / 2).

Format

```
typedef struct {
    u_long      dl_primitive;
    u_long      dl_remote_win;
} dl_hp_set_remote_win_req_t;
```

Parameters

dl_primitive

DL_HP_SET_REMOTE_WIN_REQ

dl_remote_win

size of the remote receive window. This value must be greater than 0 and less than 128.

State

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and DL_PROV_RESET_PENDING.

New State

The resulting state is unchanged.

Response

If the specified dl_remote_win is valid and the primitive was issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_SYSERR

If the specified dl_remote_win is invalid, then a DL_ERROR_ACK with dl_errno set to DL_SYSERR and dl_unix_errno set to EINVAL is returned.

DL_HP_CLEAR_STATS_REQ

Requests the DLS provider to zero the mem_fails, queue_fails, ack_to_cnt, p_to_cnt, rej_to_cnt, busy_to_cnt, i_pkts_in, i_pkts_in_oos, i_pkts_in_drop, i_pkts_out, i_pkts_retrans, s_pkts_in, s_pkts_out, u_pkts_in, u_pkts_out, bad_pkts, max_retry_cnt, remote_busy_cnt, and hw_req_fails statistics which are reported in the DL_HP_INFO_ACK primitive.

Format

```
typedef struct {
    u_long     dl_primitive;
} dl_hp_clear_stats_req_t;
```

Parameters

dl_primitive

DL_HP_CLEAR_STATS_REQ

State

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and DL_PROV_RESET_PENDING.

New State

The resulting state is unchanged.

Response

If the primitive is issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_HP_SET_LOCAL_BUSY_REQ

Requests that the DLS provider inform the remote system that the local system is busy and cannot accept new data packets.

Format

```
typedef struct {
    u_long      dl_primitive;
} dl_hp_set_local_busy_req_t;
```

Parameters

dl_primitive

DL_HP_SET_LOCAL_BUSY_REQ

State

The message is valid in state IDLE.

New State

The resulting state is unchanged.

Response

If the primitive is issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_HP_CLEAR_LOCAL_BUSY_REQ

Requests that the DLS provider inform the remote system that the local system is no longer busy and is again able to accept new data packets.

Format

```
typedef struct {
    u_long      dl_primitive;
} dl_hp_clear_local_busy_req_t;
```

Parameters

dl_primitive

DL_HP_CLEAR_LOCAL_BUSY_REQ

State

The message is valid in the states DL_IDLE, DL_DATAXFER, DL_OUTCON_PENDING, DL_INCON_PENDING, DL_USER_RESET_PENDING, and DL_PROV_RESET_PENDING after a DL_HP_SET_LOCAL_BUSY_REQ message.

New State

The resulting state is unchanged.

Response

If the primitive is issued from a valid state, the DLS provider responds with a DL_OK_ACK. Otherwise a DL_ERROR_ACK is returned.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_CONNECT_REQ

Requests the DLS provider to establish a data link connection with a remote DLS user.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong       dl_primitive;
      ulong       dl_dest_addr_length;
      ulong       dl_dest_addr_offset;
      ulong       dl_qos_length;
      ulong       dl_qos_offset;
      ulong       dl_growth;
} dl_connect_req_t;
```

Parameters

dl_primitive

DL_CONNECT_REQ

dl_dest_addr_length

length of the DLSAP address that identifies the DLS user with whom a connection is to be established.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

dl_qos_length

length of the quality of service (QOS) parameter values desired by the DLS user initiating a connection.

dl_qos_offset

offset from the beginning of the M_PROTO message block where the quality of service parameters begin.

dl_growth

defines a growth field for future enhancements to this primitive. Its value must be set to zero.

State

The primitive is valid in state DL_IDLE.

New State

The resulting state is DL_OUTCON_PENDING.

Response

There is no immediate response to the connect request. However, if the connect request is accepted by the called DLS user, DL_CONNECT_CON is sent to the calling DLS user, resulting in state DL_DATAXFER.

If the request is erroneous, DL_ERROR_ACK is returned and the resulting state is unchanged.

Reasons for Failure

DL_BADADDR

The destination DLSAP address was in an incorrect format or contained invalid information.

DL_BADQOSPARAM

The quality of service parameters contained invalid values.

DL_BADQOSTYPE

The quality of service structure type was not supported by the DLS provider.

DL_ACCESS

The DLS user did not have proper permission to use the responding stream.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_CONNECT_IND

Conveys to the local DLS user that a remote (calling) DLS user wishes to establish a data link connection.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong       dl_primitive;
      ulong       dl_correlation;
      ulong       dl_called_addr_length;
      ulong       dl_called_addr_offset;
      ulong       dl_calling_addr_length;
      ulong       dl_calling_addr_offset;
      ulong       dl_qos_length;
      ulong       dl_qos_offset;
      ulong       dl_growth;
} dl_connect_ind_t;
```

Parameters

dl_primitive

DL_CONNECT_IND

dl_correlation

correlation number to be used by the DLS user to associate this message with the DL_CONNECT_RES, DL_DISCONNECT_REQ, or DL_DISCONNECT_IND that is to follow.

dl_called_addr_length

length of the address of the DLSAP for which this DL_CONNECT_IND primitive is intended.

dl_called_addr_offset

offset from the beginning of the M_PROTO message block where the called DLSAP address begins.

dl_calling_addr_length

length of the address of the DLSAP from which the DL_CONNECT_REQ primitive was sent.

dl_calling_addr_offset

offset from the beginning of the M_PROTO message block where the calling DLSAP address begins.

dl_qos_length

length of quality of service parameter values desired by the calling DLS user.

dl_qos_offset

offset from the beginning of the M_PROTO message block where the quality of service parameters begin.

dl_growth

growth field for future enhancements to this primitive. Its value will be set to zero.

State

The message is valid in state DL_IDLE, or state DL_INCON_PENDING when the maximum number of outstanding DL_CONNECT_IND primitives has not been reached on this stream.

New State

The resulting state is DL_INCON_PENDING, regardless of the current state.

Response

The DLS user must eventually send either DL_CONNECT_RES to accept the connect request or DL_DISCONNECT_REQ to reject the connect request. In either case, the responding message must convey the correlation number received in the DL_CONNECT_IND. The DLS provider will use the correlation number to identify the connect request to which the DLS user is responding.

DL_CONNECT_RES

Directs the DLS provider to accept a connect request from a remote (calling) DLS user on a designated stream. The DLS user may accept the connection on the same stream where the connect indication arrived, or on a different stream that has been previously bound.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong        dl_primitive;
      ulong        dl_correlation;
      ulong        dl_resp_token;
      ulong        dl_qos_length;
      ulong        dl_qos_offset;
      ulong        dl_growth;
} dl_connect_res_t;
```

Parameters

dl_primitive

DL_CONNECT_RES

dl_correlation

correlation number that was received with the DL_CONNECT_IND associated with the connection request. The DLS provider will use the correlation number to identify the connect indication to which the DLS user is responding.

dl_resp_token

if non-zero, the token associated with the responding stream on which the DLS provider is to establish the connection; this stream must be attached to a PPA and bound to a DLSAP.

dl_qos_length

length of the quality of service parameter. This should be the same parameter specified in the DL_CONNECT_IND.

dl_qos_offset

offset from the beginning of the M_PROTO message block where the quality of service parameters begin.

dl_growth

growth field for future enhancements to this primitive. Its value will be set to zero.

State

The primitive is valid in state DL_INCON_PENDING.

New State

The resulting state is DL_CONN_RES_PENDING.

Response

If the connect response is successful, DL_OK_ACK is sent to the DLS user. If no outstanding connect indications remain, the resulting state for the current stream is DL_IDLE; otherwise, it remains DL_INCON_PENDING. For the responding stream (designated by the parameter dl_res_token), the resulting state is DL_DATAXFER. If the current stream and responding stream are the same, the resulting state of that stream is DL_DATAXFER. These streams may only be the same when the response corresponds to the only outstanding connect indication.

If the request fails, DL_ERROR_ACK is returned on the stream where the DL_CONNECT_RES primitive was received, and the resulting state of that stream and the responding stream is unchanged.

Reasons for Failure

DL_BADTOKEN

The token for the responding stream was not associated with a currently open stream.

DL_BADQOSPARAM

The quality of service parameters contained invalid values.

DL_BADQOSTYPE

The quality of service structure type was not supported by the DLS provider.

DL_BADCORR

The correlation number specified in this primitive did not correspond to a pending connect indication.

DL_ACCESS

The DLS user did not have proper permission to use the responding stream.

DL_OUTSTATE

The primitive was issued from an invalid state, or the responding stream was not in a valid state for establishing a connection.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_PENDING

Current stream and responding stream is the same and there is more than one outstanding connect indication.

DL_CONNECT_CON

Informs the local DLS user that the requested data link connection has been established.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong       dl_primitive;
      ulong       dl_resp_addr_length
      ulong       dl_resp_addr_offset
      ulong       dl_qos_lenth;
      ulong       dl_qos_offset;
      ulong       dl_growth;
} dl_connect_con_t;
```

Parameters

dl_primitive

DL_CONNECT_CON

dl_resp_addr_length

length of the address of the responding DLSAP associated with the newly established data link connection.

dl_resp_addr_offset

offset from the beginning of the M_PROTO message block where the responding DLSAP address begins.

dl_qos_length

length of the quality of service parameter the DLS user selected when issued the DL_CONNECT_REQ.

dl_qos_offset

offset from the beginning of the M_PROTO message block where the quality of service parameter begin.

dl_growth

growth field for future enhancements to this primitive. Its value will be set to zero.

State

The message is valid in state DL_OUTCON_PENDING.

New State

The resulting state is DL_DATAXFER.

DL_TOKEN_REQ

Requests that a connection response token be assigned to the stream and returned to the DLS user. This token can be supplied in the DL_CONNECT_RES primitive to indicate the stream on which a connection will be established.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong      dl_primitive;
} dl_token_req_t;
```

Parameters

dl_primitive

DL_TOKEN_REQ

State

The message is valid in any state in which a local acknowledgement is not pending, as described in Appendix B, Allowable Sequence of DLPI Primitives, of the DLPI 2.0 specification.

New State

The resulting state is unchanged.

Response

The DLS provider responds to the information request with a DL_TOKEN_ACK.

DL_TOKEN_ACK

This message is sent in response to DL_TOKEN_REQ; it conveys the connection response token assigned to the stream.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong        dl_primitive;
      ulong        dl_token;
} dl_token_ack_t;
```

Parameters

dl_primitive

DL_TOKEN_ACK

dl_token

connection response token associated with the stream. This value must be a non-zero value. The DLS provider will generate a token value for each stream upon receipt of the first DL_TOKEN_REQ primitive issued on that stream. The same token value will be returned in response to all subsequent DL_TOKEN_REQ primitives issued on a stream.

State

The message is valid in any state in response to a DL_TOKEN_REQ.

New State

The resulting state is unchanged.

DL_DATA_REQ

Conveys a complete DLS Data Unit (DLSDU) from the DLS user to the DLS provider for transmission over the data link connection.

Format

The message consists of one or more M_DATA message blocks containing at least one byte of data.

State

The message is valid in state DL_DATAXFER. If it is received in state DL_IDLE or DL_PROV_RESET_PENDING, it should be discarded without generating an error.

New State

The resulting state is unchanged.

Response

If the request is valid, no response is generated. If the request is erroneous, a STREAMS M_ERROR message should be issued to the DLS user specifying an errno value of EPROTO. This action should be interpreted as a fatal, unrecoverable, protocol error. A request is considered erroneous under the following conditions.

- The primitive was issued from an invalid state. If the request is issued in state DL_IDLE or DL_PROV_RESET_PENDING, however, it is silently discarded with no fatal error generated.
- The amount of data in the current DLSDU is not within the DLS provider's acceptable bounds as specified by dl_min_sdu and dl_max_sdu in the DL_INFO_ACK.

DL_DATA_IND

Conveys a DLSDU from the DLS provider to the DLS user.

Format

The message consists of one or more M_DATA message blocks containing at least one byte of data.

State

The message is valid in state DL_DATAXFER.

New State

The resulting state is unchanged.

DL_DISCONNECT_REQ

Requests the DLS provider to disconnect an active data link connection or one that was in the process of activation, either outgoing or incoming, as a result of an earlier DL_CONNECT_IND or DL_CONNECT_REQ. If an incoming DL_CONNECT_IND is being refused, the correlation number associated with that connect indication must be supplied. The message indicates the reason for the disconnection.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong       dl_primitive;
      ulong       dl_reason;
      ulong       dl_correlation;
} dl_disconnect_req_t;
```

Parameters

dl_primitive

DL_DISCONNECT_REQ

dl_reason

reason for the disconnection.

DL_DISC_NORMAL_CONDITION: normal release of a data link connection.

DL_DISC_ABNORMAL_CONDITION: abnormal release of a data link connection.

DL_CONREJ_PERMANENT_COND: a permanent condition caused the rejection of a connect request.

DL_CONREJ_TRANSIENT_COND: a transient condition caused the rejection of a connect request.

DL_UNSPECIFIED: reason unspecified

dl_correlation

if non-zero, conveys the correlation number that was contained in the DL_CONNECT_IND being rejected. This value permits the DLS provider to associate the primitive with the proper DL_CONNECT_IND when rejecting an incoming connection. If disconnect request is releasing a connection that is already established, or is aborting a previously sent DL_CONNECT_REQ, the value of dl_correlation should be zero.

State

The message is valid in any of the states: DL_DATAXFER, DL_INCON_PENDING, DL_OUTCON_PENDING, DL_PROV_RESET_PENDING, DL_USER_RESET_PENDING.

New State

The resulting state is one of the disconnect pending states, as defined in Appendix B, Allowable Sequence of DLPI Primitives, of the DLPI 2.0 specification.

Response

If the disconnect is successful, DL_OK_ACK is sent to the DLS user resulting in state DL_IDLE.

If the request fails, DL_ERROR_ACK is returned, and the resulting state is unchanged.

Reasons for Failure

DL_BADCORR

The correlation number specified in this primitive did not correspond to a pending connect indication.

DL_OUTSTATE

The primitive was issued from an invalid state.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_DISCONNECT_IND

Informs the DLS user that the data link connection on this stream has been disconnected, or that a pending connection (either DL_CONNECT_REQ or DL_CONNECT_IND) has been aborted. This primitive indicates the origin and the cause of the disconnect.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong       dl_primitive;
      ulong       dl_originator;
      ulong       dl_reason;
      ulong       dl_correlation;
} dl_disconnect_ind_t;
```

Parameters

dl_primitive

DL_DISCONNECT_IND

dl_originator

whether the disconnect was DLS user or DLS provider originated (DL_USER or DL_PROVIDER, respectively).

dl_reason

the reason for the disconnection:

DL_DISC_PERMANENT_CONDITION: connection release due to permanent connection.

DL_DISC_TRANSIENT_CONDITION: connection released due to transient connection.

DL_CONREJ_DEST_UNKOWN: unknown destination for connect request.

DL_CONREJ_DEST_UNREACH_PERMANENT: could not reach destination for connect request - permanent condition.

DL_CONREJ_DEST_UNREACH_TRANSIENT: could not reach destination for connect request - transient condition.

DL_CONREJ_QOS_UNAVAIL_PERMANENT: requested quality of service parameters permanently unavailable during connection establishment.

DL_CONREJ_QOS_UNAVAIL_TRANSIENT: requested quality of service parameters temporarily unavailable during connection establishment.

DL_UNSPECIFIED: reason unspecified

dl_correlation

if non-zero, the correlation number that was contained in the DL_CONNECT_IND that is being aborted. This value permits the DLS user to associate the message with the proper DL_CONNECT_IND. If the disconnect indication is indicating the release of a connection that is already established, or is indicating the rejection of a previously sent DL_CONNECT_REQ, the value of dl_correlation should be zero.

State

The message is valid in any of the states: DL_DATAXFER, DL_INCON_PENDING, DL_OUTCON_PENDING, DL_PROV_RESET_PENDING, DL_USER_RESET_PENDING.

New State

The resulting state is DL_IDLE.

DL_RESET_REQ

Requests that the DLS provider initiate the re-synchronization of a data link connection. This service is abortive, so no guarantee of delivery can be assumed about data that is in transit when the reset request is initiated.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
     ulong       dl_primitive;
} dl_reset_req_t;
```

Parameters

dl_primitive

DL_RESET_REQ

State

The message is valid in state DL_DATAXFER.

New State

The resulting state is DL_USER_RESET_PENDING.

Response

If the disconnect is successful, DL_OK_ACK is sent to the DLS user resulting in state DL_IDLE.

If the request fails, DL_ERROR_ACK is returned, and the resulting state is unchanged.

Reasons for Failure

DL_OUTSTATE

The primitive was issued from an invalid state.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_RESET_IND

Informs the DLS user that either the remote DLS user is re-synchronizing the data link connection, or the DLS provider is reporting loss of data for which it can not recover. The indication conveys the reason for the reset.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong       dl_primitive;
      ulong       dl_originator;
      ulong       dl_reason;
} dl_reset_ind_t;
```

Parameters

dl_primitive

DL_RESET_REQ

dl_originator

whether the reset was originated by the DLS user or DLS provider (DL_USER or DL_ PROVIDER, respectively).

dl_reason

reason for the reset:

DL_RESET_FLOW_CONTROL: indicates flow control congestion

DL_RESET_LINK_ERROR: indicates a data link error situation

DL_RESET_RESYNCH: indicates a request for re-synchronization of a data link connection.

State

The message is valid in state DL_DATAXFER.

New State

The resulting state is DL_PROV_RESET_PENDING.

Response

The DLS user should issue a DL_RESET_RES primitive to continue the resynchronization procedure.

DL_RESET_RES

Directs the DLS provider to complete re-synchronizing the data link connection.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong      dl_primitive;
} dl_reset_res_t;
```

Parameters

dl_primitive

DL_RESET_RES

State

The primitive is valid in state DL_PROV_RESET_PENDING.

New State

The resulting state is DL_RESET_RES_PENDING.

Response

If the reset response is successful, DL_OK_ACK is sent to the DLS user resulting in state DL_DATAXFER.

If the reset response is erroneous, DL_ERROR_ACK is returned, and the resulting state is unchanged.

Reasons for Failure

DL_OUTSTATE

The primitive was issued from an invalid state.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_RESET_CON

Informs the reset-initiating DLS user that the reset has completed.

Format

The message consists of one M_PROTO message block containing the structure shown below.

```
typedef struct {
      ulong       dl_primitive;
} dl_reset_con_t;
```

Parameters

dl_primitive

DL_RESET_CON

State

The message is valid in state DL_USER_RESET_PENDING.

New State

The resulting state is DL_DATAXFER.

Primitives to Handle XID and TEST Operations

This section describes the primitives used for XID and TEST operations.

DL_TEST_REQ

Conveys the TEST command DLSDU from the DLS user to the DLS provider for transmission to a peer DLS provider.

Format

The message consists of one M_PROTO message block, followed by zero or more M_DATA blocks containing zero or more bytes of data. The message structure is as follows:

```
typedef  struct  {
      ulong     dl_primitive;
      ulong     dl_flag;
      ulong     dl_dest_addr_length;
      ulong     dl_dest_addr_offset;
}  dl_test_req_t;
```

Parameters

dl_primitive

DL_TEST_REQ

dl_flag

flag values for the request as follows:

DL_POLL_FINAL indicates if the poll/final bit is set.

dl_dest_addr_length

length of the DLSAP address of the destination DLS user. If the destination user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

State

The message is valid in states DL_IDLE and DL_DATAXFER.

New State

The resulting state is unchanged.

Response

On an invalid TEST command request, a DL_ERROR_ACK is issued to the user. If the DLS provider receives a response from the remote side, a DL_TEST_CON is issued to the DLS user. It is recommended that the DLS user use a timeout procedure to recover from a situation when there is no response from the peer DLS user.

Reasons for Failure

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_BADADDR

DLSAP address information was invalid or was in an incorrect format.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_NOTSUPPORTED

Primitive is known but not supported by the DLS provider.

DL_TESTAUTO

Previous bind request specified automatic handling of TEST responses.

DL_UNSUPPORTED

Requested service not supplied by provider.

DL_TEST_IND

Conveys the TEST indication DLSDU from the DLS provider to the DLS user.

Format

The message consists of one M_PROTO message block, followed by zero or more M_DATA blocks containing zero or more bytes of data. The message structure is as follows:

```
typedef  struct  {
      ulong     dl_primitive;
      ulong     dl_flag;
      ulong     dl_dest_addr_length;
      ulong     dl_dest_addr_offset;
      ulong     dl_src_addr_length;
      ulong     dl_src_addr_offset;
 }  dl_test_ind_t;
```

Parameters

dl_primitive

DL_TEST_IND

dl_flag

flag values associated with the received TEST frame:

DL_POLL_FINAL indicates if the poll/final bit is set.

dl_dest_addr_length

length of the DLSAP address of the destination DLS user. If the destination user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

dl_src_addr_length

length of the source DLSAP address. If the source user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_src_addr_offset

offset from the beginning of the M_PROTO message block where the source DLSAP address begins.

State

The message is valid in states DL_IDLE and DL_DATAXFER.

New State

The resulting state is unchanged.

Response

The DLS user must respond with a DL_TEST_RES.

DL_TEST_RES

Conveys the TEST response DLSDU from the DLS user to the DLS provider in response to a DL_TEST_IND.

Format

The message consists of one M_PROTO message block, followed by zero or more M_DATA blocks containing zero or more bytes of data. The message structure is as follows:

```
typedef  struct  {
      ulong     dl_primitive;
      ulong     dl_flag;
      ulong     dl_dest_addr_length;
      ulong     dl_dest_addr_offset;
 }  dl_test_res_t;
```

Parameters

dl_primitive

DL_TEST_RES

dl_flag

flag values for the response as follows:

DL_POLL_FINAL indicates if the poll/final bit is set.

dl_dest_addr_length

length of the DLSAP address of the destination DLS user. If the destination user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

State

The message is valid in states DL_IDLE and DL_DATAXFER.

New State

The resulting state is unchanged.

DL_TEST_CON

Conveys the TEST response DLSDU from the DLS provider to the DLS user in response to a DL_TEST_REQ.

Format

The message consists of one M_PROTO message block, followed by zero or more M_DATA blocks containing zero or more bytes of data. The message structure is as follows:

```
typedef  struct  {
      ulong     dl_primitive;
      ulong     dl_flag;
      ulong     dl_dest_addr_length;
      ulong     dl_dest_addr_offset;
      ulong     dl_src_addr_length;
      ulong     dl_src_addr_offset;
 }  dl_test_con_t;
```

Parameters

dl_primitive

DL_TEST_CON

dl_flag

flag values for the request as follows:

DL_POLL_FINAL indicates if the poll/final bit is set.

dl_dest_addr_length

length of the DLSAP address of the destination DLS user. If the destination user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

dl_src_addr_length

length of the source DLSAP address. If the source user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_src_addr_offset

offset from the beginning of the M_PROTO message block where the source DLSAP address begins.

State

The message is valid in states DL_IDLE and DL_DATAXFER.

New State

The resulting state is unchanged.

DL_XID_REQ

Conveys one XID DLSDU from the DLS user to the DLS provider for transmission to a peer DLS user.

Format

The message consists of one M_PROTO message block, followed by zero or more M_DATA blocks containing zero or more bytes of data. The message structure is as follows:

```
typedef  struct  {
      ulong     dl_primitive;
      ulong     dl_flag;
      ulong     dl_dest_addr_length;
      ulong     dl_dest_addr_offset;
 }  dl_xid_req_t;
```

Parameters

dl_primitive

DL_XID_REQ

dl_flag

flag values for the response as follows:

DL_POLL_FINAL indicates status of the poll/final bit in the xid frame.

dl_dest_addr_length

length of the DLSAP address of the destination DLS user. If the destination user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

State

The message is valid in states DL_IDLE and DL_DATAXFER.

New State

The resulting state is unchanged.

Response

On an invalid XID request, a DL_ERROR_ACK is issued to the user. If the remote side responds to the XID request, a DL_XID_CON will be sent to the user. It is recommended that the DLS user use a timeout procedure on an XID_REQ. The timeout may be used if the remote side does not respond to the XID request.

Reasons for Failure

DL_BADDATA

The amount of data in the current DLSDU exceeded the DLS provider's DLSDU limit.

DL_XIDAUTO

Previous bind request specified provider would handle XID.

DL_OUTSTATE

Primitive was issued from an invalid state.

DL_BADADDR

The DLSAP address information was invalid or was in an incorrect format.

DL_SYSERR

A system error has occurred and the UNIX system error is indicated in the DL_ERROR_ACK.

DL_NOTSUPPORTED

Primitive is known but not supported by the DLS provider.

DL_XID_IND

Conveys an XID DLSDU from the DLS provider to the DLS user.

Format

The message consists of one M_PROTO message block, followed by zero or more M_DATA blocks containing zero or more bytes of data. The message structure is as follows:

```
typedef  struct  {
      ulong     dl_primitive;
      ulong     dl_flag;
      ulong     dl_dest_addr_length;
      ulong     dl_dest_addr_offset;
      ulong     dl_src_addr_length;
      ulong     dl_src_addr_offset;
 }  dl_xid_ind_t;
```

Parameters

dl_primitive

DL_XID_IND

dl_flag

flag values associated with the received XID frame:

DL_POLL_FINAL indicates if the received xid frame had the poll/final bit set.

dl_dest_addr_length

length of the DLSAP address of the destination DLS user. If the destination user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

dl_src_addr_length

length of the source DLSAP address. If the source user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_src_addr_offset

offset from the beginning of the M_PROTO message block where the source DLSAP address begins.

State

The message is valid in states DL_IDLE and DL_DATAXFER.

New State

The resulting state is unchanged.

Response

The DLS user must respond with a DL_XID_RES.

DL_XID_RES

Conveys an XID DLSDU from the DLS user to the DLS provider in response to a DL_XID_IND.

Format

The message consists of one M_PROTO message block, followed by zero or more M_DATA blocks containing zero or more bytes of data. The message structure is as follows:

```
typedef  struct  {
      ulong     dl_primitive;
      ulong     dl_flag;
      ulong     dl_dest_addr_length;
      ulong     dl_dest_addr_offset;
  }  dl_xid_res_t;
```

Parameters

dl_primitive

DL_XID_RES

dl_flag

flag values associated with the received XID frame:

DL_POLL_FINAL

dl_dest_addr_length

length of the DLSAP address of the destination DLS user. If the destination user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

State

The message is valid in states DL_IDLE and DL_DATAXFER.

New State

The resulting state is unchanged.

DL_XID_CON

Conveys an XID DLSDU from the DLS provider to the DLS user in response to a DL_XID_REQ.

Format

The message consists of one M_PROTO message block, followed by zero or more M_DATA blocks containing zero or more bytes of data. The message structure is as follows:

```
typedef  struct  {
      ulong     dl_primitive;
      ulong     dl_flag;
      ulong     dl_dest_addr_length;
      ulong     dl_dest_addr_offset;
      ulong     dl_src_addr_length;
      ulong     dl_src_addr_offset;
 }  dl_xid_con_t;
```

Parameters

dl_primitive

DL_XID_CON

dl_flag

flag values associated with the received XID frame:

DL_POLL_FINAL

dl_dest_addr_length

length of the DLSAP address of the destination DLS user. If the destination user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_dest_addr_offset

offset from the beginning of the M_PROTO message block where the destination DLSAP address begins.

dl_src_addr_length

length of the source DLSAP address. If the source user is implemented using DLPI, this address is the full DLSAP address returned on the DL_BIND_ACK.

dl_src_addr_offset

offset from the beginning of the M_PROTO message block where the source DLSAP address begins.

State

The message is valid in states DL_IDLE and DL_DATAXFER.

New State

The resulting state is unchanged.

DLPI States

Table 2-1 describes the states associated with DLPI. It presents the state name used in the state transition table, the corresponding DLPI state name used throughout this specification, a brief description of the state, and an indication of whether the state is valid for connection-oriented data link service (DL_CODLS), connectionless data link service (DL_CLDLS), acknowledged connectionless data link service (ACLDLS) or all.

Table 2-1 DLPI States

State	DLPI State	Description	Service Type
0) UNATTACHED	DL_UNATTACHED	Stream opened but PPA not attached	ALL
1) ATTACH PEND	DL_ATTACH_ PENDING	The DLS user is waiting for an acknowledgment of a DL_ATTACH_REQ	ALL
2) DETACH PEND	DL_DETACH_ PENDING	The DLS user is waiting for an acknowledgment of a DL_DETACH_REQ	ALL
3) UNBOUND	DL_UNBOUND	Stream is attached but not bound to a DLSAP	ALL
4) BIND PEND	DL_BIND_PENDING	The DLS user is waiting for an acknowledgment of a DL_BIND_REQ	ALL
5) UNBIND PEND	DL_UNBIND_ PENDING	The DLS user is waiting for an acknowledgment of a DL_UNBIND_REQ	ALL
6) IDLE	DL_IDLE	The stream is bound and activated for use - connection establishment or connectionless data transfer may take place	ALL

State	DLPI State	Description	Service Type
7) UDQOS PEND	DL_UDQOS_ PENDING	The DLS user is waiting for an acknowledgment of a DL_UDQOS_REQ	DL_CLDLS
8) OUTCON PEND	DL_OUTCON_ PENDING	An outgoing connection is pending - the DLS user is waiting for a DL_CONNECT_CON	DL_CODLS
9) INCON PEND	DL_INCON_ PENDING	An incoming connection is pending - the DLS provider is waiting for a DL_CONNECT_RES	DL_CODLS
10) CONN_RES PEND	DL_CONN_RES_ PENDING	The DLS user is waiting for an acknowledgment of a DL_CONNECT_RES	DL_CODLS
11) DATAXFER	DL_DATAXFER	Connection-mode data transfer may take place	DL_CODLS
12) USER RESET PEND	DL_USER_RESET_ PENDING	A user-initiated reset is pending - the DLS user is waiting for a DL_RESET_CON	DL_CODLS
13) PROV RESET PEND	DL_PROV_RESET_ PENDING	A provider-initiated reset is pending - the DLS provider is waiting for a DL_RESET_RES	DL_CODLS
14) RESET_RES PEND	DL_RESET_RES_ PENDING	The DLS user is waiting for an acknowledgment of a DL_RESET_RES	DL_CODLS
15) DISCON 8 PEND	DL_DISCON8_ ENDING	The DLS user is waiting for an acknowledgment of a DL_DISCONNECT_REQ issued from the DL_OUTCON_PENDING state	DL_CODLS

State	DLPI State	Description	Service Type
16) DISCON 9 PEND	DL_DISCON9_ PENDING	The DLS user is waiting for an acknowledgement of a DL_DISCONNECT_REQ Issued from the DL_INCON_PENDING state.	DL_CODLS
17) DISCON 11 PEND	DL_DISCON11_ PENDING	The DLS user is waiting for an acknowledgment of a DL_DISCONNECT_REQ issued from the DL_DATAXFER state	DL_CODLS
18) DISCON 12 PEND	DL_DISCON12_ PENDING	The DLS user is waiting for an acknowledgment of a DL_DISCONNECT_REQ issued from the DL_USER_RESET_ PENDING state	DL_CODLS
19) DISCON 13 PEND	DL_DISCON13_ PENDING	The DLS user is waiting for an acknowledgment of a DL_DISCONNECT_REQ issued from the DL_PROV_RESET_ PENDING state	DL_CODLS
20) SUBS_BIND PEND	DL_SUBS_BIND_ PND	The DLS user is waiting for an acknowledgment of a DL_SUBS_BIND_REQ	ALL
21) SUBS_ UNBIND PEND	DL_SUBS_UNBIND_ PND	The DLS user is waiting for an acknowledgment of a DL_SUBS_UNBIND_REQ	ALL

The following rules apply to the maintenance of DLPI state:

- The DLS provider is responsible for keeping a record of the state of the interface as viewed by the DLS user, to be returned in the DL_INFO_ACK.

- The DLS provider may never generate a primitive that places the interface out of state.
- If the DLS provider generates a STREAMS M_ERROR message upstream, it should free any further primitives processed by its write side put or service procedure.
- The close of a stream is considered an abortive action by the DLS user, and may be executed from any state. The DLS provider must issue appropriate indications to the remote DLS user when a close occurs. For example, if the DLPI state is DL_DATAXFER, a DL_DISCONNECT_IND should be sent to the remote DLS user. The DLS provider should free any resources associated with that stream and reset the stream to its unopened condition.

The following points clarify the state transition table.

- If the DLS provider supports connection-mode service, the value of the outcnt state variable must be initialized to zero for each stream when that stream is first opened.
- The initial and final state for a style 2 DLS provider is DL_UNATTACHED. However, because a style 1 DLS provider implicitly attaches a PPA to a stream when it is opened, the initial and final DLPI state for a style 1 provider is DL_UNBOUND. The DLS user should not issue DL_ATTACH_REQ or DL_DETACH_REQ primitives to a style 1 DLS provider.
- A DLS provider may have multiple connect indications outstanding (i.e. the DLS user has not responded to them) at one time. As the state transition table points out, the stream on which those indications are outstanding will remain in the DL_INCON_PENDING state until the DLS provider receives a response for all indications.
- The DLPI state associated with a given stream may be transferred to another stream only when the DL_CONNECT_RES primitive indicates this behavior. In this case, the responding stream (where the connection will be established) must be in the DL_IDLE state.
- The labeling of the states DL_PROV_RESET_PENDING and DL_USER_RESET_PENDING indicate the party that started the local interaction, and does not necessarily indicate the originator of the reset procedure.

- A DL_DATA_REQ primitive received by the DLS provider in the state DL_PROV_RESET_PENDING (i.e. after a DL_RESET_IND has been passed to the DLS user) or the state DL_IDLE (i.e. after a data link connection has been released) should be discarded by the DLS provider.
- A DL_DATA_IND primitive received by the DLS user after the user has issued a DL_RESET_REQ should be discarded.

To ensure accurate processing of DLPI primitives, the DLS provider must adhere to the following rules concerning the receipt and generation of STREAMS M_FLUSH messages during various state transitions.

- The DLS provider must be ready to receive M_FLUSH messages from upstream and flush its queues as specified in the message.
- The DLS provider must issue an M_FLUSH message upstream to flush both the read and write queues after receiving a successful DL_UNBIND_REQ primitive but before issuing the DL_OK_ACK.
- If an incoming disconnect occurs when the interface is in the DL_DATAXFER, DL_USER_RESET_PENDING, or DL_PROV_RESET_PENDING state, the DLS provider must send up an M_FLUSH message to flush both the read and write queues before sending up a DL_DISCONNECT_IND.
- If a DL_DISCONNECT_REQ is issued in the DL_DATAXFER, DL_USER_RESET_PENDING, or DL_PROV_RESET_PENDING states, the DLS provider must issue an M_FLUSH message upstream to flush both the read and write queues after receiving the successful DL_DISCONNECT_REQ but before issuing the DL_OK_ACK.
- If a reset occurs when the interface is in the DL_DATAXFER or DL_USER_RESET_PENDING state, the DLS provider must send up an M_FLUSH message to flush both the read and write queues before sending up a DL_RESET_IND or DL_RESET_CON.

A Sample Programs

This appendix contains sample programs for connection, connectionless, and raw modes.

Connection Mode

```
/*****************************************************************************
  (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1992. ALL RIGHTS
  RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
  REPRODUCED, OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
  THE PRIOR WRITTEN CONSENT OF HEWLETT PACKARD COMPANY
*****************************************************************************/

/*****************************************************************************
        This program demonstrates data transfer over a connection oriented
        DLPI stream.  It also demonstrates connection handoff.
*****************************************************************************/

#include <stdio.h>
#include <fcntl.h>
#include <memory.h>
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/dlpi.h>
#include <sys/dlpi_ext.h>

#define SEND_SAP        0x80            /* sending SAP */
#define RECV_SAP        0x82            /* receiving SAP */

/*****************************************************************************
        global areas for sending and receiving messages
*****************************************************************************/
#define AREA_SIZE  5000   /* bytes; big enough for largest possible msg */

#define LONG_AREA_SIZE  (AREA_SIZE / sizeof(u_long)) /* AREA_SIZE / 4 */

/* these are u_long arrays instead of u_char to insure proper alignment */
u_long  ctrl_area[LONG_AREA_SIZE];      /* for control messages */
u_long  data_area[LONG_AREA_SIZE];      /* for data messages */

struct strbuf ctrl_buf = {
        AREA_SIZE,              /* maxlen = AREA_SIZE */
        0,                      /* len gets filled in for each message */
        ctrl_area               /* buf = control area */
};

struct strbuf data_buf = {
        AREA_SIZE,              /* maxlen = AREA_SIZE */
        0,                      /* len gets filled in for each message */
        data_area               /* buf = data area */
};

/*****************************************************************************
        get the next message from a stream; get_msg() returns one of the
        following defines
*****************************************************************************/
```

```
#define GOT_CTRL        1       /* message has only a control part */
#define GOT_DATA        2       /* message has only a data part */
#define GOT_BOTH        3       /* message has both control and data parts */

int
get_msg(fd)
        int     fd;             /* file descriptor */
{
        int     flags = 0;      /* 0 ---> get any available message */
        int     result = 0;     /* return value */
        /*
        zero first byte of control area so the caller can call check_ctrl
        without checking the get_msg return value; if there was only data
        in the message and the user was expecting control or control + data,
        then when he calls check_ctrl it will compare the expected primitive
        zero and print information about the primitive that it got.
        */
        ctrl_area[0] = 0;

        /* call getmsg and check for an error */
        if(getmsg(fd, &ctrl_buf, &data_buf, &flags) < 0) {
                printf("error:  getmsg failed, errno = %d\n", errno);
                exit(1);
        }
        if(ctrl_buf.len > 0) {
                result |= GOT_CTRL;
        }
        if(data_buf.len > 0) {
                result |= GOT_DATA;
        }
        return(result);
}
/*************************************************************************
        check that control message is the expected message
*************************************************************************/
void
check_ctrl(ex_prim)
        int     ex_prim;        /* the expected primitive */
{
        dl_error_ack_t  *err_ack = (dl_error_ack_t *)ctrl_area;

        /* did we get the expected primitive? */
        if(err_ack->dl_primitive != ex_prim) {
                /* did we get a control part */
                if(ctrl_buf.len) {
                        /* yup; is it an ERROR_ACK? */
                        if(err_ack->dl_primitive == DL_ERROR_ACK) {
                                /* yup; format the ERROR_ACK info */
                                printf("error:  expected primitive 0x%02x, ",
                                                ex_prim);
                                printf("got DL_ERROR_ACK\n");
                                printf("   dl_error_primitive = 0x%02x\n",
                                                err_ack->dl_error_primitive);
                                printf("   dl_errno = 0x%02x\n",
                                                err_ack->dl_errno);
                                printf("   dl_unix_errno = %d\n",
```

```
                                        err_ack->dl_unix_errno);
                        exit(1);
                } else {
                        /*
                        didn't get an ERROR_ACK either; print whatever
                        primitive we did get
                         */
                        printf("error:  expected primitive 0x%02x, ",
                                        ex_prim);
                        printf("got primitive 0x%02x\n",
                                        err_ack->dl_primitive);
                        exit(1);
                }
        } else {
                /* no control; did we get data? */
                if(data_buf.len) {
                      /* tell user we only got data */
                      printf("error:  check_ctrl found only data\n");
                      exit(1);
                } else {
                        /*
                    no message???; well, it was probably an
                    interrupted system call
                    */
                    printf("error:  check_ctrl found no message\n");
                    exit(1);
                }
        }
    }
}

/*************************************************************************
        put a message consisting of only a data part on a stream
*************************************************************************/
void
put_data(fd, length)
        int     fd;             /* file descriptor */
        int     length;         /* length of data message */
{
        /* set the len field in the strbuf structure */
        data_buf.len = length;

        /* call putmsg and check for an error */
        if(putmsg(fd, 0, &data_buf, 0) < 0) {
                printf("error:  put_data putmsg failed, errno = %d\n", errno);
                exit(1);
        }
}
```

```
/*****************************************************************************
        put a message consisting of only a control part on a stream
*****************************************************************************/
void
put_ctrl(fd, length, pri)
        int     fd;         /* file descriptor */
        int     length;     /* length of control message */
        int     pri;        /* priority of message: either 0 or RS_HIPRI */
{
        /* set the len field in the strbuf structure */
        ctrl_buf.len = length;

        /* call putmsg and check for an error */
        if(putmsg(fd, &ctrl_buf, 0, pri) < 0) {
                printf("error:  put_ctrl putmsg failed, errno = %d\n", errno);
                exit(1);
        }
}

/*****************************************************************************
        put a message consisting of both a control part and a control part
        on a stream
*****************************************************************************/
void
put_both(fd, ctrl_length, data_length, pri)
        int     fd;          /* file descriptor */
        int     ctrl_length; /* length of control part */
        int     data_length; /* length of data part */
        int     pri;         /* priority of message: either 0 or RS_HIPRI */
{
        /* set the len fields in the strbuf structures */
        ctrl_buf.len = ctrl_length;
        data_buf.len = data_length;

        /* call putmsg and check for an error */
        if(putmsg(fd, &ctrl_buf, &data_buf, pri) < 0) {
                printf("error:  put_both putmsg failed, errno = %d\n", errno);
                exit(1);
        }
}

/*****************************************************************************
        print a string followed by a DLSAP
*****************************************************************************/
void
print_dlsap(string, dlsap, dlsap_len)
        char    *string;        /* label */
        u_char  *dlsap;         /* the DLSAP */
```

```
        int     dlsap_len;      /* length of dlsap */
{
        int     i;

        printf("%s0x", string);
        for(i = 0; i < dlsap_len; i++) {
                printf("%02x", dlsap[i]);
        }
        printf("\n");
}
/*************************************************************************
    open the DLPI cloneable device file, get a list of available PPAs,
    and attach to the first PPA; returns a file descriptor for the stream
*************************************************************************/
int
attach() {
        int     fd;             /* file descriptor */
        int     ppa;            /* PPA to attach to */
        dl_hp_ppa_req_t *ppa_req = (dl_attach_req_t *)ctrl_area;
        dl_hp_ppa_ack_t *ppa_ack = (dl_hp_ppa_ack_t *)ctrl_area;
        dl_hp_ppa_info_t *ppa_info;
        dl_attach_req_t *attach_req = (dl_attach_req_t *)ctrl_area;
        char *mac_name;

        /* open the device file */
        if((fd = open("/dev/dlpi", O_RDWR)) == -1) {
                printf("error:  open failed, errno = %d\n", errno);
                exit(1);
        }

        /*
        find a PPA to attach to;  we assume that the first PPA on the
        remote is on the same media as the first local PPA
        */
        /* send a PPA_REQ and wait for the PPA_ACK */
        ppa_req->dl_primitive = DL_HP_PPA_REQ;
        put_ctrl(fd, sizeof(dl_hp_ppa_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_HP_PPA_ACK);
        /* make sure we found at least one PPA */
        if(ppa_ack->dl_length == 0) {
                printf("error:  no PPAs available\n");
                exit(1);
        }
        /* examine the first PPA */
        ppa_info = (dl_hp_ppa_info_t *)((u_char *)ctrl_area +
                        ppa_ack->dl_offset);
        ppa = ppa_info->dl_ppa;
        switch(ppa_info->dl_mac_type) {
                case DL_CSMACD:
                case DL_ETHER:
                        mac_name = "Ethernet";
                        break;
                case DL_TPR:
                        mac_name = "Token Ring";
                        break;
```

```
                case DL_FDDI:
                        mac_name = "FDDI";
                        break;
                default:
                        printf("error:  unknown MAC type in ppa_info\n");
                        exit(1);
        }
        printf("attaching to %s media on PPA %d\n", mac_name, ppa);

        /*
        fill in ATTACH_REQ with the PPA we found, send the ATTACH_REQ,
        and wait for the OK_ACK
        */
        attach_req->dl_primitive = DL_ATTACH_REQ;
        attach_req->dl_ppa = ppa;
        put_ctrl(fd, sizeof(dl_attach_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_OK_ACK);

        /* return the file descriptor for the stream to the caller */
        return(fd);
}

/*****************************************************************************
        bind to a sap with a specified service mode and max_conind;
        returns the local DLSAP and its length
*****************************************************************************/
void
bind(fd, sap, max_conind, service_mode, dlsap, dlsap_len)
        int     fd;             /* file descriptor */
        int     sap;            /* 802.2 SAP to bind on */
        int     max_conind;     /* max # of connect indications to accept */
        int     service_mode;   /* either DL_CODLS or DL_CLDLS */
        u_char  *dlsap;         /* return DLSAP */
        int     *dlsap_len;     /* return length of dlsap */
{
        dl_bind_req_t   *bind_req = (dl_bind_req_t *)ctrl_area;
        dl_bind_ack_t   *bind_ack = (dl_bind_ack_t *)ctrl_area;
        u_char          *dlsap_addr;

        /* fill in the BIND_REQ */
        bind_req->dl_primitive = DL_BIND_REQ;
        bind_req->dl_sap = sap;
        bind_req->dl_max_conind = max_conind;
        bind_req->dl_service_mode = service_mode;
        bind_req->dl_conn_mgmt = 0;   /* conn_mgmt is NOT supported */
        bind_req->dl_xidtest_flg = 0; /* user will handle TEST & XID pkts */
```

```
        /* send the BIND_REQ and wait for the OK_ACK */
        put_ctrl(fd, sizeof(dl_bind_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_BIND_ACK);

        /* return the DLSAP to the caller */
        *dlsap_len = bind_ack->dl_addr_length;
        dlsap_addr = (u_char *)ctrl_area + bind_ack->dl_addr_offset;
        memcpy(dlsap, dlsap_addr, *dlsap_len);
}

/*************************************************************************
        unbind, detach, and close
*************************************************************************/
void
cleanup(fd)
        int     fd;             /* file descriptor */
{
        dl_unbind_req_t *unbind_req = (dl_unbind_req_t *)ctrl_area;
        dl_detach_req_t *detach_req = (dl_detach_req_t *)ctrl_area;

        /* unbind */
        unbind_req->dl_primitive = DL_UNBIND_REQ;
        put_ctrl(fd, sizeof(dl_unbind_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_OK_ACK);

        /* detach */
        detach_req->dl_primitive = DL_DETACH_REQ;
        put_ctrl(fd, sizeof(dl_detach_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_OK_ACK);

        /* close */
        close(fd);
}

/*************************************************************************
        send a connect request to a DLSAP
*************************************************************************/
void
connect_req(fd, dlsap, dlsap_len)
        int     fd;             /* file descriptor */
        u_char  *dlsap;         /* DLSAP to connect with */
        int     dlsap_len;      /* length of dlsap */
{
        dl_connect_req_t        *con_req = (dl_connect_req_t *)ctrl_area;
        dl_connect_res_t        *con_res = (dl_connect_res_t *)ctrl_area;
        u_char  *tdlsap;

        /* fill in the connect request */
        con_req->dl_primitive = DL_CONNECT_REQ;
        con_req->dl_dest_addr_length = dlsap_len;
        con_req->dl_dest_addr_offset = sizeof(dl_connect_req_t);
        /* QOS is not supported; these fields must be set to zero */
        con_req->dl_qos_length = 0;
        con_req->dl_qos_offset = 0;
```

```
        con_req->dl_growth = 0;

        /* copy in the dlsap */
        tdlsap = (u_char *)ctrl_area + con_req->dl_dest_addr_offset;
        memcpy(tdlsap, dlsap, dlsap_len);

        /* send the connect request */
        print_dlsap("sending CONNECT_REQ to DLSAP ", dlsap, dlsap_len);
        put_ctrl(fd, sizeof(dl_connect_req_t) + dlsap_len, 0);
}

/*************************************************************************
        get a connection response token for a stream; returns the token
*************************************************************************/
u_long
get_token(fd)
        int     fd;             /* file descriptor */
{
        dl_token_req_t  *tok_req = (dl_token_req_t *)ctrl_area;
        dl_token_ack_t  *tok_ack = (dl_token_ack_t *)ctrl_area;

        /*
        Send down a token request.  Note that unlike most of the other
        messages this one is a PCPROTO message so we call put_ctrl with
        RS_HIPRI instead of zero.
        */
        tok_req->dl_primitive = DL_TOKEN_REQ;
        put_ctrl(fd, sizeof(dl_token_req_t), RS_HIPRI);

        /* wait for the token ack */
        get_msg(fd);
        check_ctrl(DL_TOKEN_ACK);

        /* return the token */
        return(tok_ack->dl_token);
}

/*************************************************************************
        get a connect indication from a stream; returns the correlation number
*************************************************************************/
u_long
connect_ind(fd)
        int     fd;             /* file descriptor */
{
        dl_connect_ind_t        *con_ind = (dl_connect_ind_t *)ctrl_area;
        u_char  *dlsap;
        int     dlsap_len;

        /* wait for the connect indication */
        get_msg(fd);
        check_ctrl(DL_CONNECT_IND);

        /* print the calling DLSAP */
        dlsap = (u_char *)ctrl_area + con_ind->dl_calling_addr_offset;
        dlsap_len = con_ind->dl_calling_addr_length;
        print_dlsap("received CONNECT_IND from DLSAP ", dlsap, dlsap_len);
```

```
        /* return the correlation number */
        return(con_ind->dl_correlation);
}

/*************************************************************************
        send a connect response with a specified correlation and token;
        wait for the OK_ACK
*************************************************************************/
void
connect_res(fd, correlation, token)
        int     fd;             /* file descriptor */
        u_long  correlation;    /* correlation number of CONNECT_IND */
                                        /* being responded to */
        u_long  token;          /* token of stream to pass connection to */
{
        dl_connect_res_t        *con_res = (dl_connect_res_t *)ctrl_area;

        /* fill in the connect response */
        con_res->dl_primitive = DL_CONNECT_RES;
        con_res->dl_correlation = correlation;
        con_res->dl_resp_token = token;
        /* QOS is not supported; these fields must be set to zero */
        con_res->dl_qos_length = 0;
        con_res->dl_qos_offset = 0;
        con_res->dl_growth = 0;

        put_ctrl(fd, sizeof(dl_connect_res_t), 0);
        get_msg(fd);
        check_ctrl(DL_OK_ACK);
}

/*************************************************************************
        send a DISCONNECT_REQ and wait for the OK_ACK
*************************************************************************/
void
disconnect_req(fd)
        int     fd;             /* file descriptor */
{
        dl_disconnect_req_t     *disc_req = (dl_disconnect_req_t *)ctrl_area;

        /* fill in the disconnect request */
        disc_req->dl_primitive = DL_DISCONNECT_REQ;
        /* this is a normal disconnect */
        disc_req->dl_reason = DL_DISC_NORMAL_CONDITION;
        /*
        Since we are not rejecting a CONNECT_IND, we set the correlation
        to zero.
```

```
        */
        disc_req->dl_correlation = 0;

        /* send the disconnect request */
        put_ctrl(fd, sizeof(dl_disconnect_req_t), 0);

        /* wait for the OK_ACK */
        get_msg(fd);
        check_ctrl(DL_OK_ACK);
}
/*************************************************************************
        main
*************************************************************************/
main() {
        int     send_fd;        /* file descriptor for sending stream */
        int     recv_c_fd;      /* fd for recv ctrl stream */
        int     recv_d_fd;      /* fd for recv data stream */
        u_char  sdlsap[20];     /* sending DLSAP */
        u_char  rcdlsap[20];    /* receiving control DLSAP */
        u_char  rddlsap[20];    /* receiving data DLSAP */
        int     sdlsap_len, rcdlsap_len, rddlsap_len;   /* DLSAP lengths */
        u_long  correlation;    /* correlation number for CONNECT_IND */
        u_long  token;          /* token for recv_d stream */
        int     i;              /* loop counter */

        /*
        We'll use three streams: a sending stream, a receiving stream bound
        with max_conind = 1 (the "control" stream), and a receiving stream
        bound with max_conind = 0 (the "data" stream). The connect indication
        will be handed off from the control stream to the data stream.
        We initially open only the sending stream and the receiving control
        stream.
        */

        /*
        First, we must open the DLPI device file, /dev/dlpi, and attach
        to a PPA.  attach will open /dev/dlpi, find the first PPA
        with the DL_HP_PPA_INFO primitive, and attach to that PPA.
        attach() returns the file descriptor for the stream.
        */
        send_fd = attach();
        recv_c_fd = attach();

        /*
        Now we must bind the streams to saps.  The bind function will
        return the local DLSAP and its length for each stream in the last
        two arguments.  Only the receiving control stream has a non-zero
        max_conind.
        */
        bind(send_fd, SEND_SAP, 0, DL_CODLS, sdlsap, &sdlsap_len);
        bind(recv_c_fd, RECV_SAP, 1, DL_CODLS, rcdlsap, &rcdlsap_len);

        /*
        Start the connection establishment process by sending a CONNECT_REQ
        from the sender to the receiver control stream.
        */
```

```
connect_req(send_fd, rcdlsap, rcdlsap_len);

/*
The receiver control stream gets a CONNECT_IND.  We need the
correlation number to relate the CONNECT_IND to the CONNECT_RES
we will send down later.
*/
correlation = connect_ind(recv_c_fd);

/*
We want to handle the actual data transfer over a dedicated
receiver stream.  Here, we attach and bind a second stream on
the receivers sap with max_conind = 0.
*/
recv_d_fd = attach();
bind(recv_d_fd, RECV_SAP, 0, DL_CODLS, rddlsap, &rddlsap_len);

/*
To pass the connection from the control stream to the data stream,
we need a token for the data stream.  get_token returns this.
*/
token = get_token(recv_d_fd);

/*
Now we do a CONNECT_RES on the control stream.  The correlation
specifies the CONNECT_IND we are responding to, and the token,
since it is non-zero, specifies the stream to which we want to
hand off the connection.
*/
connect_res(recv_c_fd, correlation, token);

/* Get the CONNECT_CON back on the senders stream */
get_msg(send_fd);
check_ctrl(DL_CONNECT_CON);
printf("connection established\n");

/*
We now have a connection established between the send_fd stream and
the recv_d_fd stream.  The recv_c_fd stream is in the IDLE state
and is ready to process another CONNECT_IND.  Since we won't be
establishing any new connections, we'll call cleanup on the
receiver control stream to unbind, detach, and close the file
descriptor.
*/
cleanup(recv_c_fd);

/* Fill in data_area with some data to send. */
for(i = 0; i < 60; i++) {
        data_area[i] = i;
}

/* Send 5 packets of data. */
for(i = 0; i < 5; i++) {
        put_data(send_fd, (i + 1) * 10);
        printf("sent %d bytes of data\n", (i + 1) * 10);
}
```

```
        /* Receive the 5 packets. */
        for(i = 0; i < 5; i++) {
                if(get_msg(recv_d_fd) != GOT_DATA) {
                        printf("error:  didn't get data\n");
                        check_ctrl(0);
                        exit(1);
                }
                printf("received %d bytes of data\n", data_buf.len);
        }

        /*
        We're finished.  Now we tear down the connection.  We'll send
        a DISCONNECT_REQ on the receiver side.
        */
        disconnect_req(recv_d_fd);

        /* and receive the DISCONNECT_IND on the sender side. */
        get_msg(send_fd);
        check_ctrl(DL_DISCONNECT_IND);

        /* And finally, we tear down the sender and receiver streams */
        cleanup(send_fd);
        cleanup(recv_d_fd);
}
```

Connectionless Mode

```
/*****************************************************************************
  (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1992. ALL RIGHTS
  RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
  REPRODUCED, OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
  THE PRIOR WRITTEN CONSENT OF HEWLETT PACKARD COMPANY
*****************************************************************************/

/*****************************************************************************
        The main part of this program is composed of two parts.
        The first part demonstrates data transfer over a connectionless
        stream with LLC SAP headers.  The second part of this program
        demonstrates data transfer over a connectionless stream with
        LLC SNAP headers.
*****************************************************************************/

#include <stdio.h>
#include <fcntl.h>
#include <memory.h>
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/dlpi.h>
#include <sys/dlpi_ext.h>

#define SEND_SAP        0x80            /* sending SAP */
#define RECV_SAP        0x82            /* receiving SAP */
#define SNAP_SAP        0xAA            /* SNAP SAP */

/*****************************************************************************
        SNAP protocol values.
*****************************************************************************/
u_char SEND_SNAP_SAP[5] = {0x50, 0x00, 0x00, 0x00, 0x00};
u_char RECV_SNAP_SAP[5] = {0x60, 0x00, 0x00, 0x00, 0x00};

/*****************************************************************************
        global areas for sending and receiving messages
*****************************************************************************/
#define AREA_SIZE       5000    /* bytes; big enough for largest possible msg */

#define LONG_AREA_SIZE  (AREA_SIZE / sizeof(u_long)) /* AREA_SIZE / 4 */

u_long  ctrl_area[LONG_AREA_SIZE];      /* for control messages */
u_long  data_area[LONG_AREA_SIZE];      /* for data messages */

struct strbuf ctrl_buf = {
        AREA_SIZE,              /* maxlen = AREA_SIZE */
        0,                      /* len gets filled in for each message */
        ctrl_area               /* buf = control area */
};

struct strbuf data_buf = {
```

```
        AREA_SIZE,              /* maxlen = AREA_SIZE */
        0,                      /* len gets filled in for each message */
        data_area               /* buf = data area */
};

/*************************************************************************
        get the next message from a stream; get_msg() returns one of the
        following defines
*************************************************************************/
#define GOT_CTRL        1     /* message has only a control part */
#define GOT_DATA        2     /* message has only a data part */
#define GOT_BOTH        3     /* message has both control and data parts */

int
get_msg(fd)
        int     fd;             /* file descriptor */
{
        int     flags = 0;      /* 0 ---> get any available message */
        int     result = 0;     /* return value */

        /*
        zero first byte of control area so the caller can call check_ctrl
        without checking the get_msg return value; if there was only data
        in the message and the user was expecting control or control + data,
        then when he calls check_ctrl it will compare the expected primitive
        zero and print information about the primitive that it got.
        */
        ctrl_area[0] = 0;

        /* call getmsg and check for an error */
        if(getmsg(fd, &ctrl_buf, &data_buf, &flags) < 0) {
                printf("error:  getmsg failed, errno = %d\n", errno);
                exit(1);
        }
        if(ctrl_buf.len > 0) {
                result |= GOT_CTRL;
        }
        if(data_buf.len > 0) {
                result |= GOT_DATA;
        }
        return(result);
}
/*************************************************************************
        check that control message is the expected message
*************************************************************************/
void
check_ctrl(ex_prim)
        int     ex_prim;        /* the expected primitive */
{
        dl_error_ack_t  *err_ack = (dl_error_ack_t *)ctrl_area;

        /* did we get the expected primitive? */
        if(err_ack->dl_primitive != ex_prim) {
                /* did we get a control part */
                if(ctrl_buf.len) {
                        /* yup; is it an ERROR_ACK? */
```

```
                        if(err_ack->dl_primitive == DL_ERROR_ACK) {
                                /* yup; format the ERROR_ACK info */
                                printf("error:  expected primitive 0x%02x, ",
                                                ex_prim);
                                printf("got DL_ERROR_ACK\n");
                                printf("   dl_error_primitive = 0x%02x\n",
                                                err_ack->dl_error_primitive);
                                printf("   dl_errno = 0x%02x\n",
                                                err_ack->dl_errno);
                                printf("   dl_unix_errno = %d\n",
                                                err_ack->dl_unix_errno);
                                exit(1);
                        } else {
                                /*
                                didn't get an ERROR_ACK either; print whatever
                                primitive we did get
                                 */
                                printf("error:  expected primitive 0x%02x, ",
                                                ex_prim);
                                printf("got primitive 0x%02x\n",
                                                err_ack->dl_primitive);
                                exit(1);
                        }
                } else {
                        /* no control; did we get data? */
                        if(data_buf.len) {
                            /* tell user we only got data */
                            printf("error:  check_ctrl found only data\n");
                            exit(1);
                        } else {
                            /*
                            no message???; well, it was probably an
                            interrupted system call
                            */
                            printf("error:  check_ctrl found no message\n");
                            exit(1);
                        }
                }
        }
}

/*********************************************************************
        put a message consisting of only a data part on a stream
*********************************************************************/
void
put_data(fd, length)
        int     fd;             /* file descriptor */
        int     length;         /* length of data message */
```

```
{
        /* set the len field in the strbuf structure */
        data_buf.len = length;

        /* call putmsg and check for an error */
        if(putmsg(fd, 0, &data_buf, 0) < 0) {
            printf("error:  put_data putmsg failed, errno = %d\n", errno);
            exit(1);
        }
}

/*****************************************************************************
        put a message consisting of only a control part on a stream
*****************************************************************************/
void
put_ctrl(fd, length, pri)
        int     fd;         /* file descriptor */
        int     length;     /* length of control message */
        int     pri;        /* priority of message: either 0 or RS_HIPRI */
{
        /* set the len field in the strbuf structure */
        ctrl_buf.len = length;

        /* call putmsg and check for an error */
        if(putmsg(fd, &ctrl_buf, 0, pri) < 0) {
            printf("error:  put_ctrl putmsg failed, errno = %d\n", errno);
            exit(1);
        }
}

/*****************************************************************************
        put a message consisting of both a control part and a control part
        on a stream
*****************************************************************************/
void
put_both(fd, ctrl_length, data_length, pri)
        int     fd;          /* file descriptor */
        int     ctrl_length; /* length of control part */
        int     data_length; /* length of data part */
        int     pri;         /* priority of message: either 0 or RS_HIPRI */
{
        /* set the len fields in the strbuf structures */
        ctrl_buf.len = ctrl_length;
        data_buf.len = data_length;

        /* call putmsg and check for an error */
        if(putmsg(fd, &ctrl_buf, &data_buf, pri) < 0) {
            printf("error:  put_both putmsg failed, errno = %d\n", errno);
            exit(1);
        }
}

/*****************************************************************************
        open the DLPI cloneable device file, get a list of available PPAs,
        and attach to the first PPA; returns a file descriptor for the stream
*****************************************************************************/
int
```

```
attach() {
        int     fd;             /* file descriptor */
        int     ppa;            /* PPA to attach to */
        dl_hp_ppa_req_t *ppa_req = (dl_attach_req_t *)ctrl_area;
        dl_hp_ppa_ack_t *ppa_ack = (dl_hp_ppa_ack_t *)ctrl_area;
        dl_hp_ppa_info_t *ppa_info;
        dl_attach_req_t *attach_req = (dl_attach_req_t *)ctrl_area;
        char *mac_name;

        /* open the device file */
        if((fd = open("/dev/dlpi", O_RDWR)) == -1) {
                printf("error:  open failed, errno = %d\n", errno);
                exit(1);
        }

        /*
        find a PPA to attach to;  we assume that the first PPA on the
        remote is on the same media as the first local PPA
        */
        /* send a PPA_REQ and wait for the PPA_ACK */
        ppa_req->dl_primitive = DL_HP_PPA_REQ;
        put_ctrl(fd, sizeof(dl_hp_ppa_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_HP_PPA_ACK);
        /* make sure we found at least one PPA */
        if(ppa_ack->dl_length == 0) {
                printf("error:  no PPAs available\n");
                exit(1);
        }
        /* examine the first PPA */
        ppa_info = (dl_hp_ppa_info_t *)((u_char *)ctrl_area +
                        ppa_ack->dl_offset);
        ppa = ppa_info->dl_ppa;
        switch(ppa_info->dl_mac_type) {
                case DL_CSMACD:
                case DL_ETHER:
                        mac_name = "Ethernet";
                        break;
                case DL_TPR:
                        mac_name = "Token Ring";
                        break;
                case DL_FDDI:
                        mac_name = "FDDI";
                        break;
                default:
                        printf("error:  unknown MAC type in ppa_info\n");
                        exit(1);
        }
        printf("attaching to %s media on PPA %d\n", mac_name, ppa);

        /*
        fill in ATTACH_REQ with the PPA we found, send the ATTACH_REQ,
        and wait for the OK_ACK
```

```
        */
        attach_req->dl_primitive = DL_ATTACH_REQ;
        attach_req->dl_ppa = ppa;
        put_ctrl(fd, sizeof(dl_attach_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_OK_ACK);

        /* return the file descriptor for the stream to the caller */
        return(fd);
}
/*************************************************************************
        bind to a sap with a specified service mode and max_conind;
        returns the local DLSAP and its length
*************************************************************************/
void
bind(fd, sap, max_conind, service_mode, dlsap, dlsap_len)
        int     fd;             /* file descriptor */
        int     sap;            /* 802.2 SAP to bind on */
        int     max_conind;     /* max # of connect indications to accept */
        int     service_mode;   /* either DL_CODLS or DL_CLDLS */
        u_char  *dlsap;         /* return DLSAP */
        int     *dlsap_len;     /* return length of dlsap */
{
        dl_bind_req_t   *bind_req = (dl_bind_req_t *)ctrl_area;
        dl_bind_ack_t   *bind_ack = (dl_bind_ack_t *)ctrl_area;
        u_char          *dlsap_addr;

        /* fill in the BIND_REQ */
        bind_req->dl_primitive = DL_BIND_REQ;
        bind_req->dl_sap = sap;
        bind_req->dl_max_conind = max_conind;
        bind_req->dl_service_mode = service_mode;
        bind_req->dl_conn_mgmt = 0;   /* conn_mgmt is NOT supported */
        bind_req->dl_xidtest_flg = 0; /* user will handle TEST & XID pkts */

        /* send the BIND_REQ and wait for the OK_ACK */
        put_ctrl(fd, sizeof(dl_bind_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_BIND_ACK);

        /* return the DLSAP to the caller */
        *dlsap_len = bind_ack->dl_addr_length;
        dlsap_addr = (u_char *)ctrl_area + bind_ack->dl_addr_offset;
        memcpy(dlsap, dlsap_addr, *dlsap_len);
}

/*************************************************************************
        bind to a SNAP sap via the DL_PEER_BIND, or DL_HIERARCHICAL_BIND
        subsequent bind class;  returns the local DLSAP and its length
*************************************************************************/
void
subs_bind(fd, snapsap, snapsap_len, subs_bind_class, dlsap, dlsap_len)
int     fd;
u_char  *snapsap;
int     subs_bind_class;
u_char  *dlsap;
```

```
int     *dlsap_len;
{
      dl_subs_bind_req_t    *subs_bind_req = (dl_subs_bind_req_t*)ctrl_area;
      dl_subs_bind_ack_t    *subs_bind_ack = (dl_subs_bind_ack_t*)ctrl_area;
      u_char                *dlsap_addr;

        /* Fill in Subsequent bind req */
        subs_bind_req->dl_primitive = DL_SUBS_BIND_REQ;
        subs_bind_req->dl_subs_sap_offset = DL_SUBS_BIND_REQ_SIZE;
        subs_bind_req->dl_subs_sap_length = snapsap_len;
        subs_bind_req->dl_subs_bind_class = subs_bind_class;
        memcpy((caddr_t)&subs_bind_req[1], snapsap, snapsap_len);

        /* send the SUBS_BIND_REQ and wait for the OK_ACK */
        put_ctrl(fd, sizeof(dl_subs_bind_req_t)+snapsap_len, 0);
        get_msg(fd);
        check_ctrl(DL_SUBS_BIND_ACK);

        /* return the DLSAP to the caller */
        *dlsap_len = subs_bind_ack->dl_subs_sap_length;
        dlsap_addr = (u_char *)ctrl_area + subs_bind_ack->dl_subs_sap_offset;
        memcpy(dlsap, dlsap_addr, *dlsap_len);

}
/*********************************************************************
        unbind, detach, and close
*********************************************************************/
void
cleanup(fd)
        int     fd;             /* file descriptor */
{
        dl_unbind_req_t *unbind_req = (dl_unbind_req_t *)ctrl_area;
        dl_detach_req_t *detach_req = (dl_detach_req_t *)ctrl_area;

        /* unbind */
        unbind_req->dl_primitive = DL_UNBIND_REQ;
        put_ctrl(fd, sizeof(dl_unbind_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_OK_ACK);

        /* detach */
        detach_req->dl_primitive = DL_DETACH_REQ;
        put_ctrl(fd, sizeof(dl_detach_req_t), 0);
        get_msg(fd);
        check_ctrl(DL_OK_ACK);

        /* close */
        close(fd);
}

/*********************************************************************
        receive a data packet;
*********************************************************************/
int
recv_data(fd)
        int     fd;             /* file descriptor */
```

```
{
        dl_unitdata_ind_t       *data_ind = (dl_unitdata_ind_t *)ctrl_area;
        char    *rdlsap;
        int     msg_res;

        msg_res = get_msg(fd);
        check_ctrl(DL_UNITDATA_IND);
        if(msg_res != GOT_BOTH) {
                printf("error:  did not receive data part of message\n");
                exit(1);
        }
        return(data_buf.len);
}

/*************************************************************************
        send a data packet; assumes data_area has already been filled in
*************************************************************************/
void
send_data(fd, rdlsap, rdlsap_len, len)
        int     fd;             /* file descriptor */
        u_char  *rdlsap;        /* remote dlsap */
        int     rdlsap_len;     /* length of rdlsap */
        int     len;            /* length of the packet to send */
{
        dl_unitdata_req_t       *data_req = (dl_unitdata_req_t *)ctrl_area;
        u_char  *out_dlsap;

        /* fill in data_req */
        data_req->dl_primitive = DL_UNITDATA_REQ;
        data_req->dl_dest_addr_length = rdlsap_len;
        data_req->dl_dest_addr_offset = sizeof(dl_unitdata_req_t);
        /* copy dlsap */
        data_req->dl_priority.dl_min = 0;
        data_req->dl_priority.dl_max = 0;
        out_dlsap = (u_char *)ctrl_area + sizeof(dl_unitdata_req_t);
        memcpy(out_dlsap, rdlsap, rdlsap_len);
        put_both(fd, sizeof(dl_unitdata_req_t) + rdlsap_len, len, 0);
}

/*************************************************************************
        print a string followed by a DLSAP
*************************************************************************/
void
print_dlsap(string, dlsap, dlsap_len)
        char    *string;                /* label */
        u_char  *dlsap;                 /* the DLSAP */
        int     dlsap_len;              /* length of dlsap */
{
        int     i;

        printf("%s", string);
        for(i = 0; i < dlsap_len; i++) {
                printf("%02x", dlsap[i]);
        }
        printf("\n");
}
```

```
/*********************************************************************
        main
*********************************************************************/
main() {
        int     send_fd, recv_fd;       /* file descriptors */
        u_char  sdlsap[20];             /* sending DLSAP */
        u_char  rdlsap[20];             /* receiving DLSAP */
        int     sdlsap_len, rdlsap_len; /* DLSAP lengths */
        int     i, j, recv_len;

        /*
        PART 1 of program.  Demonstrate connectionless data transfer with
        LLC SAP header.
        */

        /*
        First, we must open the DLPI device file, /dev/dlpi, and attach
        to a PPA.  attach() will open /dev/dlpi, find the first PPA
        with the DL_HP_PPA_INFO primitive, and attach to that PPA.
        attach() returns the file descriptor for the stream.  Here we
        do an attach for each file descriptor.
        */
        send_fd = attach();
        recv_fd = attach();

        /*
        Now we have to bind to a IEEESAP. We will ask for connectionless data
        link service with the DL_CLDLS service mode.  Since we are
        connectionless, we will not have any incoming connections so we
        set max_conind to 0.  bind() will return our local DLSAP and its
        length in the last two arguments we pass to it.
        */
        bind(send_fd, SEND_SAP, 0, DL_CLDLS, sdlsap, &sdlsap_len);
        bind(recv_fd, RECV_SAP, 0, DL_CLDLS, rdlsap, &rdlsap_len);

        /* print the DLSAPs we got back from the binds */
        print_dlsap("sending DLSAP   = ", sdlsap, sdlsap_len);
        print_dlsap("receiving DLSAP = ", rdlsap, rdlsap_len);

        /*
        Time to send some data.  We'll send 5 data packets in sequence.
        */
        for(i = 0; i < 5; i++) {
                /* send (i+1)*10 data bytes with the first byte = i */
                data_area[0] = i;
                /* Initialize data area */
                for (j = 1; j < (i+1)*10; j++)
                        data_area[j] = "a";
                print_dlsap("sending data to ",rdlsap, rdlsap_len);
                send_data(send_fd, rdlsap, rdlsap_len, (i + 1) * 10);
                /* receive the data packet */
                recv_len = recv_data(recv_fd);
                printf("received %d bytes, first word = %d\n", recv_len,
                                data_area[0]);
        }
```

```
/*
We're finished with PART 1. Now call cleanup to unbind, then detach,
then close the device file.
*/
cleanup(send_fd);
cleanup(recv_fd);

/*
PART 2 of program.  Demonstrate connectionless data transfer with
LLC SNAP SAP header.
*/

/*
As demonstrated in the first part of this program we must first
open the DLPI device file, /dev/dlpi, and attach to a PPA.
*/
send_fd = attach();
recv_fd = attach();

/*
The first method for binding a SNAP protocol value (which is
demonstrated below) requires the user to first bind the SNAP
SAP 0xAA, then issue a subsequent bind with class DL_HIERARCHICAL_BIND
with the 5 bytes of SNAP information.

The second method (which is not demonstrated in this program) is
to bind any supported protocol value (see section 5) and then issue
a subsequent bind with class DL_PEER_BIND.  The data area of the
subsequent bind should include 6 bytes of data, the first byte being
the SNAP SAP 0xAA followed by 5 bytes of SNAP information.
*/
bind(send_fd, SNAP_SAP, 0, DL_CLDLS, sdlsap, &sdlsap_len);
bind(recv_fd, SNAP_SAP, 0, DL_CLDLS, rdlsap, &rdlsap_len);

/*
Now we must complete the binding of the SNAP protocol value
with the subsequent bind request and a subsequent bind class
of DL_HIERARCHICAL_BIND.
*/
subs_bind(send_fd, SEND_SNAP_SAP, 5, DL_HIERARCHICAL_BIND, sdlsap,
                                        &sdlsap_len);
subs_bind(recv_fd, RECV_SNAP_SAP, 5, DL_HIERARCHICAL_BIND, rdlsap,
                                        &rdlsap_len);
/* print the DLSAPs we got back from the binds */
print_dlsap(”sending DLSAP   = ”, sdlsap, sdlsap_len);
print_dlsap(”receiving DLSAP = ”, rdlsap, rdlsap_len);

/*
Time to send some data.  We'll send 5 data packets in sequence.
*/
for(i = 0; i < 5; i++) {
        /* send (i+1)*10 data bytes with the first byte = i */
        data_area[0] = i;
        /* Initialize data area */
        for (j = 1; j < (i+1)*10; j++)
                data_area[j] = ”a”;
```

```
                print_dlsap("sending data to ",rdlsap, rdlsap_len);
                send_data(send_fd, rdlsap, rdlsap_len, (i + 1) * 10);
                /* receive the data packet */
                recv_len = recv_data(recv_fd);
                printf("received %d bytes, first word = %d\n", recv_len,
                                data_area[0]);
        }

        /*
        We're finished.  Now call cleanup to unbind, then detach,
        then close the device file.
        */
        cleanup(send_fd);
        cleanup(recv_fd);
}
```

Raw Mode

```
/*
 * (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1993. ALL RIGHTS
 * RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
 * REPRODUCED, OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
 * THE PRIOR WRITTEN CONSENT OF HEWLETT PACKARD COMPANY
 */

/*************************************************************
        The program demonstrates RAW mode data transfer over an
        802.3 interface.
*************************************************************/

#define PPA 1
#define FRAME_LEN 1500 /* max message size is 1514;MAC+LLC+data */
#define SEQ_OFFSET 100
#define INSAP 22
#define OUTSAP 24

#define OUTER_LOOPS 10
#define INNER_LOOPS 25

#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <math.h>
#include <ctype.h>

#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/poll.h>
#include <sys/dlpi.h>
#include <sys/dlpi_ext.h>
#include <netinet/if_ieee.h>

#define bcopy(source, destination, length) memcpy(destination, source, length)
#define ETHER_HLEN 14

char tag[80];

#define AREA_SZ 5000    /*-=-* buffer length in bytes *-=-*/

u_long  ctl_area[AREA_SZ];
u_long  dat_area[AREA_SZ];

struct strbuf ctl = {AREA_SZ, 0, ctl_area};
struct strbuf dat = {AREA_SZ, 0, dat_area};

#define GOT_CTRL    1
```

```
#define GOT_DATA    2
#define GOT_BOTH    3
#define GOT_INTR    4

/*-=-* get a message from a stream; return type of message *-=-*/
int
get_msg(fd)
    int     fd;
{
    int     flags = 0;
    int     res, ret;

    ctl_area[0] = 0;
    dat_area[0] = 0;
    ret = 0;

    res = getmsg(fd, &ctl, &dat, &flags);

    if(res < 0) {
        if(errno == EINTR) {
            return(GOT_INTR);
        } else {
            printf("%s,get_msg:  getmsg failed, errno = %d\n", tag, errno);
            exit(1);
        }
    }
    if(ctl.len > 0) {
        ret |= GOT_CTRL;
    }
    if(dat.len > 0) {
        ret |= GOT_DATA;
    }
    return(ret);
}

/*-=-* verify that dl_primitive in ctl_area = prim *-=-*/
int
check_ctrl(prim)
    int     prim;
{
    dl_error_ack_t  *err_ack = (dl_error_ack_t *)ctl_area;

    if(err_ack->dl_primitive != prim) {
        if(err_ack->dl_primitive == DL_ERROR_ACK) {
            printf("%s,check_ctrl:  got DL_ERROR_ACK\n",tag);
            printf("   dl_error_primitive = 0x%02x\n",
                    err_ack->dl_error_primitive);
```

```
            printf("   dl_errno = 0x%02x\n", err_ack->dl_errno);
            printf("   dl_unix_errno = %d\n", err_ack->dl_unix_errno);
            exit(1);
        } else {
            printf("%s,check_ctrl:  expected primitive 0x%02x", tag, prim);
            printf(", got primitive 0x%02x\n", err_ack->dl_primitive);
            exit(1);
        }
    }
}

/*-=-* put a control message on a stream *-=-*/
void
put_ctrl(fd, len, pri)
    int     fd, len, pri;
{
    ctl.len = len;

    if(putmsg(fd, &ctl, 0, pri) < 0) {
        printf("%s,put_ctrl:  putmsg failed, errno = %d\n", tag, errno);
        exit(1);
    }
}

/*-=-* put a control + data message on a stream *-=-*/
void
put_both(fd, clen, dlen, pri)
    int     fd, clen, dlen, pri;
{
    ctl.len = clen;
    dat.len = dlen;

    if(putmsg(fd, &ctl, &dat, pri) < 0) {
        printf("%s,put_both:  putmsg failed, errno = %d\n", tag, errno);
        exit(1);
    }
}

/*-=-* open file descriptor and attach *-=-*/
int
dl_open(ppa)
    int             ppa;
{
    int             fd;
    dl_attach_req_t *attach_req = (dl_attach_req_t *)ctl_area;

    if((fd = open("/dev/dlpi", O_RDWR)) == -1) {
        printf("%s,dl_open:  open failed, errno = %d\n",tag, errno);
        exit(1);
    }

    attach_req->dl_primitive = DL_ATTACH_REQ;
    attach_req->dl_ppa = ppa;

    put_ctrl(fd, sizeof(dl_attach_req_t), 0);
    get_msg(fd);
    check_ctrl(DL_OK_ACK);
```

```
    return(fd);
}

/*-=-* send DL_BIND_REQ *-=-*/
void
dl_bind(fd, sap, addr)
    int     fd, sap;
    u_char *addr;
{
    dl_bind_req_t   *bind_req = (dl_bind_req_t *)ctl_area;
    dl_bind_ack_t   *bind_ack = (dl_bind_ack_t *)ctl_area;

    bind_req->dl_primitive = DL_BIND_REQ;
    bind_req->dl_sap = sap;
    bind_req->dl_max_conind = 1;
    bind_req->dl_service_mode = DL_HP_RAWDLS;
    bind_req->dl_conn_mgmt = 0;
    bind_req->dl_xidtest_flg = 0;

    put_ctrl(fd, sizeof(dl_bind_req_t), 0);
    get_msg(fd);
    check_ctrl(DL_BIND_ACK);

    bcopy((u_char *)bind_ack + bind_ack->dl_addr_offset, addr,
                bind_ack->dl_addr_length);
}

void xxx();

void
main(argc, argv)
    int     argc;
    char    *argv[];
{
    int infd, outfd;
    struct pollfd pinfo;
    int i, j, inseq;
    u_char addr[25];
    struct ieee8023_hdr *mac_hdr = (struct ieee8023_hdr *)dat_area;
    struct ieee8022_hdr *llc_hdr;
    dl_hp_rawdata_req_t *rawdat_req = (dl_hp_rawdata_req_t *)ctl_area;
    dl_hp_rawdata_ind_t *rawdat_ind = (dl_hp_rawdata_ind_t *)ctl_area;
    dl_error_ack_t *err_ack = (dl_error_ack_t *)ctl_area;

    /* MAC header size is 14 bytes */
    llc_hdr = (struct ieee8022_hdr *)&((u_char *)dat_area)[14];

    if(!(infd = dl_open(PPA))) {
        printf("error: open failed\n");
        exit(1);
    }
    if(!(outfd = dl_open(PPA))) {
        printf("error: open failed\n");
        exit(1);
    }
    dl_bind(infd, INSAP, addr);
```

```
    dl_bind(outfd, OUTSAP, addr);

    pinfo.fd = outfd;
    pinfo.events = POLLIN | POLLPRI;
    pinfo.revents = 0;

    for(i = 0; i < OUTER_LOOPS; i++) {
        for(j = 0; j < INNER_LOOPS; j++) {
            bcopy(addr, mac_hdr->destaddr, 6);
            /* card will stuff in source addr
             * The ieee header length does not include the
             * ethernet MAC header.
             */
            mac_hdr->length = FRAME_LEN - ETHER_HLEN;
            llc_hdr->dsap = INSAP;
            llc_hdr->ssap = OUTSAP;
            llc_hdr->ctrl = IEEECTRL_DEF;
            sprintf(&dat_area[SEQ_OFFSET], "%d", i * INNER_LOOPS + j);
            rawdat_req->dl_primitive = DL_HP_RAWDATA_REQ;
            put_both(outfd, sizeof(dl_hp_rawdata_req_t), FRAME_LEN, 0);
            printf("+");
            fflush(stdout);
            if(poll(&pinfo, 1, 0)) {
                get_msg(outfd);
                check_ctrl(DL_ERROR_ACK);
                if(err_ack->dl_error_primitive != DL_HP_RAWDATA_REQ ||
                        err_ack->dl_errno != DL_SYSERR ||
                        err_ack->dl_unix_errno != ENOBUFS) {
                    check_ctrl(0);
                } else {
                    /* re-send same pkt */
                    printf("\nENOBUFS\n");
                    j--;
                }
            }
        }

        for(j = 0; j < INNER_LOOPS; j++) {
            get_msg(infd);
            printf("-");
            fflush(stdout);
            check_ctrl(DL_HP_RAWDATA_IND);
            if(dat.len != FRAME_LEN) {
                printf("\nlength error:  expected %d, got %d\n",
                        FRAME_LEN, dat.len);
            }
            inseq = strtol(&dat_area[SEQ_OFFSET], 0, 0);
            if(inseq != (i * INNER_LOOPS + j)) {
                printf("\nseq error:  expected %d, got %d\n",
                        i * INNER_LOOPS + j, inseq);
            }
        }
    }
    printf("\n");
}
```

Glossary

Called DLS user The DLS user in connection mode that processes requests for connections from other DLS users.

Calling DLS user The DLS user in connection mode that initiates the establishment of a data link connection.

Communication endpoint

The local communication channel between a DLS user and DLS provider.

Connection establishment

The phase in connection mode that enables two DLS users to create a data link connections between them.

Connectionless mode A mode of transfer in which data is passed from one user to another in self-contained units with no logical relationship required among the units.

Connection mode A circuit-oriented mode of transfer in which data is passed from one user to another over an established connection in a sequenced manner.

Connection release The phase in connection mode that terminates a previously established data link connection.

Data link service data unit A grouping of DLS user data whose boundaries are preserved from one end of a data link connection to the other.

Data transfer The phase in connection and connectionless mode that supports the transfer of data between two DLS users.

DLSAP A point at which a DLS user attached itself to a DLS provider to access data link services.

DLSAP address An identifier used to differentiate and locate specific DLS user access points to a DLS provider.

DLS provider The data link layer protocol that provides the services of the Data Link

Provider Interface.

DLS user The user-level application or user-level or kernel-level protocol that accessess the services of the data link layer.

Local management The phase in connection and connectionless modes in which a DLS user initiates a stream and binds a DLSAP to the stream. Primitives in this phase generate local operations only.

PPA The point at which a system attaches itself to a physical communications medium.

PPA identifier An identifier of a particular physical medium over which communication transpires.

Index

D

DL_ATTACH_REQ, 61
DL_BIND_ACK, 65
DL_BIND_REQ, 63
DL_CONNECT_CON, 116
DL_CONNECT_IND, 111
DL_CONNECT_REQ, 110
DL_CONNECT_RES, 113
DL_DATA_IND, 119
DL_DATA_REQ, 118
DL_DETACH_REQ, 62
DL_DISABMULTI_REQ, 72
DL_DISCONNECT_IND, 121
DL_DISCONNECT_REQ, 119
DL_ENABMULTI_REQ, 71
DL_ERROR_ACK, 77
DL_GET_STATISTICS_ACK, 82
DL_GET_STATISTICS_REQ, 82
DL_HP_CLEAR_LOCAL_BUSY_REQ, 109
DL_HP_CLEAR_STATS_REQ, 107
DL_HP_INFO_ACK, 95
DL_HP_INFO_REQ, 95
DL_HP_MULTICAST_LIST_ACK, 84
DL_HP_MULTICAST_LIST_REQ, 83
DL_HP_PPA_ACK, 53
DL_HP_PPA_REQ, 52
DL_HP_RAWDATA_IND, 92
DL_HP_RAWDATA_REQ, 91
DL_HP_SET_ACK_THRESH_REQ, 104
DL_HP_SET_ACK_TO_REQ, 99
DL_HP_SET_BUSY_TO_REQ, 101
DL_HP_SET_LOCAL_BUSY_REQ, 108
DL_HP_SET_LOCAL_WIN_REQ, 105
DL_HP_SET_MAX_RETRIES_REQ, 103
DL_HP_SET_P_TO_REQ, 100
DL_HP_SET_REJ_TO_REQ, 101
DL_HP_SET_REMOTE_WIN_REQ, 106
DL_HP_SET_SEND_ACK_TO_REQ, 103
DL_INFO_ACK, 56
DL_INFO_REQ, 56
DL_OK_ACK, 77
DL_PHYS_ADDR_ACK, 80
DL_PHYS_ADDR_REQ, 78
DL_PROMISCOFF_REQ, 75
DL_PROMISCON_REQ, 74
DL_RESET_CON, 125
DL_RESET_IND, 123
DL_RESET_REQ, 123
DL_RESET_RES, 124
DL_SET_PHYS_ADDR_REQ, 80
DL_SUBS_BIND_ACK, 69
DL_SUBS_BIND_REQ, 67
DL_SUBS_UNBIND_REQ, 70
DL_TEST_CON, 131
DL_TEST_IND, 128
DL_TEST_REQ, 127
DL_TEST_RES, 130
DL_TOKEN_ACK, 117
DL_TOKEN_REQ, 117
DL_UDERROR_IND, 89
DL_UNBIND_REQ, 67
DL_UNITDATA_IND, 88
DL_UNITDATA_REQ, 86
DL_XID_CON, 136
DL_XID_IND, 133
DL_XID_REQ, 132
DL_XID_RES, 135
DLPI
  binding, 32
  connection handoff, 36
  device file format, 16
  DLSAP addressing, 23
  features, 15
  header files, 16
  PPA format, 22
  unsupported features, 15
DLPI extensions
  connection-oriented, 94