

# *CAE Specification*

## **Data Link Provider Interface (DLPI)**

*The Open Group*



© February 1997, *The Open Group*

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

The base document for this specification was Copyright 1991 by UNIX International. See Acknowledgements.

CAE Specification

Data Link Provider Interface (DLPI)

ISBN: 1-85912-196-9

Document Number: C614

Published in the U.K. by The Open Group, February 1997.

Any comments relating to the material contained in this document may be submitted to:

The Open Group  
Apex Plaza  
Forbury Road  
Reading  
Berkshire, RG1 1AX  
United Kingdom

or by Electronic Mail to:

[OGSpecs@opengroup.org](mailto:OGSpecs@opengroup.org)

# Contents

<b>Chapter 1</b>	<b>Introduction.....</b>	<b>1</b>
1.1	Overview .....	1
1.2	Model of the Data Link Layer .....	2
1.2.1	Model of the Service Interface.....	2
1.3	Modes of Communication .....	4
1.3.1	Connection-mode Service .....	4
1.3.1.1	Local Management .....	4
1.3.1.2	Connection Establishment .....	4
1.3.1.3	Data Transfer.....	5
1.3.1.4	Connection Release .....	5
1.3.2	Connectionless-mode Service .....	5
1.3.3	Acknowledged Connectionless-mode Service .....	5
1.4	DLPI Addressing.....	6
1.4.1	Physical Attachment Identification.....	6
1.4.2	Data Link User Identification .....	7
1.5	Connection Management Stream .....	8
<b>Chapter 2</b>	<b>DLPI Services .....</b>	<b>9</b>
2.1	Management Services .....	11
2.1.1	Information Reporting Service.....	11
2.1.2	Attach Service .....	11
2.1.3	Bind Service.....	12
2.2	Connection-mode Services .....	15
2.2.1	Connection Establishment Service.....	15
2.2.1.1	Normal Connection Establishment.....	15
2.2.1.2	Connection Establishment Rejections .....	16
2.2.2	Data Transfer Service .....	18
2.2.3	Connection Release Service .....	19
2.2.4	Reset Service .....	20
2.3	Connectionless-mode Services .....	23
2.3.1	Connectionless Data Transfer Service .....	23
2.3.2	QOS Management Service .....	23
2.3.3	Error Reporting Service .....	24
2.3.4	XID and TEST Service .....	24
2.4	Acknowledged Connectionless-mode Services .....	26
2.4.1	Acknowledged Connectionless-mode Data Transfer Services .....	26
2.4.2	QOS Management Service .....	27
2.4.3	Error Reporting Service .....	27
2.5	Example .....	28

<b>Chapter 3</b>	<b>DLPI Primitives</b> .....	<b>29</b>
3.1	Local Management Service Primitives .....	31
3.1.1	PPA Initialization/De-initialization .....	31
3.1.2	List of Local Management Service Primitives .....	32
	<i>DL_INFO_REQ</i> .....	33
	<i>DL_INFO_ACK</i> .....	34
	<i>DL_ATTACH_REQ</i> .....	39
	<i>DL_DETACH_REQ</i> .....	41
	<i>DL_BIND_REQ</i> .....	42
	<i>DL_BIND_ACK</i> .....	46
	<i>DL_UNBIND_REQ</i> .....	48
	<i>DL_SUBS_BIND_REQ</i> .....	49
	<i>DL_SUBS_BIND_ACK</i> .....	51
	<i>DL_SUBS_UNBIND_REQ</i> .....	52
	<i>DL_ENABMULTI_REQ</i> .....	53
	<i>DL_DISABMULTI_REQ</i> .....	54
	<i>DL_PROMISCON_REQ</i> .....	55
	<i>DL_PROMISCOFF_REQ</i> .....	56
	<i>DL_OK_ACK</i> .....	57
	<i>DL_ERROR_ACK</i> .....	58
3.2	Connection-mode Service Primitives .....	59
3.2.1	Scope.....	59
3.2.2	Multi-threaded Connection Establishment .....	59
3.2.3	List of Connection-mode Service Primitives.....	60
	<i>DL_CONNECT_REQ</i> .....	61
	<i>DL_CONNECT_IND</i> .....	63
	<i>DL_CONNECT_RES</i> .....	65
	<i>DL_CONNECT_CON</i> .....	67
	<i>DL_TOKEN_REQ</i> .....	68
	<i>DL_TOKEN_ACK</i> .....	69
	<i>DL_DATA_REQ</i> .....	70
	<i>DL_DATA_IND</i> .....	71
	<i>DL_DISCONNECT_REQ</i> .....	72
	<i>DL_DISCONNECT_IND</i> .....	74
	<i>DL_RESET_REQ</i> .....	76
	<i>DL_RESET_IND</i> .....	77
	<i>DL_RESET_RES</i> .....	78
	<i>DL_RESET_CON</i> .....	79
3.3	Connectionless-mode Service Primitives .....	80
3.3.1	Scope.....	80
3.3.2	List of Connectionless-mode Service Primitives.....	80
	<i>DL_UNITDATA_REQ</i> .....	81
	<i>DL_UNITDATA_IND</i> .....	83
	<i>DL_UDERROR_IND</i> .....	84
	<i>DL_UDQOS_REQ</i> .....	85
3.4	Primitives to handle XID and TEST operations .....	86
3.4.1	Scope.....	86
3.4.2	List of Primitives Handling XID and TEST operations.....	86

	<i>DL_TEST_REQ</i> .....	87
	<i>DL_TEST_IND</i> .....	89
	<i>DL_TEST_RES</i> .....	90
	<i>DL_TEST_CON</i> .....	91
	<i>DL_XID_REQ</i> .....	92
	<i>DL_XID_IND</i> .....	94
	<i>DL_XID_RES</i> .....	95
	<i>DL_XID_CON</i> .....	96
3.5	Acknowledged Connectionless-mode Service Primitives .....	97
3.5.1	Scope .....	97
3.5.2	List of Acknowledged Connectionless-mode Service Primitives..	97
	<i>DL_DATA_ACK_REQ</i> .....	98
	<i>DL_DATA_ACK_IND</i> .....	100
	<i>DL_DATA_ACK_STATUS_IND</i> .....	102
	<i>DL_REPLY_REQ</i> .....	103
	<i>DL_REPLY_IND</i> .....	105
	<i>DL_REPLY_STATUS_IND</i> .....	106
	<i>DL_REPLY_UPDATE_REQ</i> .....	108
	<i>DL_REPLY_UPDATE_STATUS_IND</i> .....	109
<b>Chapter 4</b>	<b>Quality of Data Link Service .....</b>	<b>111</b>
4.1	Characteristics .....	111
4.2	Overview of Quality of Service .....	111
4.2.1	Connection-mode Service .....	111
4.2.2	QOS for Connectionless/Acknowledged Connectionless .....	112
4.3	QOS Parameter Definitions .....	113
4.3.1	Throughput .....	113
4.3.2	Transit Delay .....	114
4.3.3	Priority .....	114
4.3.4	Protection .....	115
4.3.5	Residual Error Rate .....	116
4.3.6	Resilience .....	116
4.4	QOS Data Structures .....	118
4.4.1	Structure DL_QOS_CO_RANGE1 .....	118
4.4.2	Structure DL_QOS_CO_SEL1 .....	119
4.4.3	Structure DL_QOS_CL_RANGE1 .....	119
4.4.4	Structure DL_QOS_CL_SEL1 .....	120
4.5	Procedures for QOS Negotiation and Selection .....	121
4.5.1	Connection-mode QOS Negotiation .....	121
4.5.2	Connectionless-mode QOS Selection .....	124
<b>Appendix A</b>	<b>Primitives for Management Services .....</b>	<b>125</b>
	<i>DL_PHYS_ADDR_REQ</i> .....	126
	<i>DL_PHYS_ADDR_ACK</i> .....	127
	<i>DL_SET_PHYS_ADDR_REQ</i> .....	128
	<i>DL_GET_STATISTICS_REQ</i> .....	129
	<i>DL_GET_STATISTICS_ACK</i> .....	130

<b>Appendix B</b>	<b>Allowable Sequence of DLPI Primitives.....</b>	<b>131</b>
B.1	DLPI States .....	132
B.2	Variables and Actions for State Transition Table .....	135
B.3	DLPI User-originated Events .....	136
B.4	DLPI Provider-Originated Events .....	137
B.5	DLPI State Transition Table .....	139
<b>Appendix C</b>	<b>Precedence of DLPI Primitives .....</b>	<b>145</b>
C.1	Overview .....	145
C.2	Write Queue Precedence .....	146
C.3	Read Queue Precedence .....	147
<b>Appendix D</b>	<b>Guidelines for Protocol-independent DLS Users.....</b>	<b>149</b>
<b>Appendix E</b>	<b>DLS Provider-Specific Information.....</b>	<b>151</b>
E.1	DLSAP Address Space.....	151
E.2	Subsequent DLSAP Addresses .....	152
E.3	PPA Access and Control.....	152
E.4	Quality of Service.....	153
E.5	DL_INFO_ACK Values.....	153
E.6	Supported Services.....	153
<b>Appendix F</b>	<b>DLPI Header File.....</b>	<b>155</b>
	<b>Glossary .....</b>	<b>175</b>
	<b>Index.....</b>	<b>177</b>

**List of Figures**

1-1	Abstract View of DLPI.....	2
1-2	Data Link Addressing Components .....	6
2-1	Information Reporting.....	11
2-2	Attaching a Stream to a Physical Line.....	11
2-3	Detaching a Stream from a Physical Line.....	11
2-4	Binding a Stream to a DLSAP.....	12
2-5	Unbinding a Stream from a DLSAP.....	13
2-6	Enabling a specific multicast address on a Stream.....	13
2-7	Disabling a specific multicast address on a Stream.....	13
2-8	Enabling promiscuous mode on a Stream.....	14
2-9	Disabling promiscuous mode on a Stream .....	14
2-10	Successful Connection Establishment.....	15
2-11	Token Retrieval.....	16
2-12	Called DLS User Rejection of Connection Establishment Attempt ....	16
2-13	DLS Provider Rejection of a Connection Establishment Attempt.....	17
2-14	Both Primitives Destroyed by Provider .....	17
2-15	DL_DISCONNECT Indication Arrives before DL_CONNECT Response is Sent.....	17

2-16	DL_DISCONNECT Indication Arrives after DL_CONNECT Response is Sent .....	18
2-17	Normal Data Transfer .....	18
2-18	DLS User-Invoked Connection Release .....	19
2-19	Simultaneous DLS User Invoked Connection Release .....	19
2-20	DLS Provider Invoked Connection Release .....	19
2-21	Simultaneous DLS User & DLS Provider Invoked Connection Release Service .....	20
2-22	DLS User-Invoked Connection Reset .....	21
2-23	Simultaneous DLS User-Invoked Connection Reset.....	21
2-24	DLS Provider-Invoked Connection Reset.....	22
2-25	Simultaneous DLS User & DLS Provider Invoked Connection Reset .....	22
2-26	Connectionless Data Transfer .....	23
2-27	QOS Service .....	23
2-28	Error Reporting Service .....	24
2-29	XID Service.....	24
2-30	TEST Service .....	25
2-31	Acknowledged Connectionless-Mode Data Unit Transmission Service .....	26
2-32	Acknowledged Connectionless-Mode Data Unit Exchange Service .....	26
2-33	Acknowledged Connectionless-Mode Reply Data Unit Preparation Service .....	27
2-34	A Connection-mode Example.....	28

**List of Tables**

2-1	Cross-Reference of DLS Services and Primitives.....	10
4-1	QOS Supported Parameters.....	113
B-1	States Associated with DLPI .....	132
B-2	DPLI State Transition Table Variables .....	135
B-3	DLPI State Transition Actions.....	135
B-4	DLPI User-oriented Events.....	136
B-5	DLPI Provider-originated Events.....	138
B-6	DLPI State Transition - Local Management Phase.....	141
B-7	DLPI State Transition - Connectionless-mode Data Transfer Phase... ..	141
B-8	DLPI State Transition - Acknowledged Connectionless-mode Data Transfer Phase .....	142
B-9	DLPI State Transition - Connection Establishment Phase .....	142
B-10	DLPI State Transition - Connection-mode Data Transfer Phase.....	143
C-1	Write Queue Precedence .....	146
C-2	Read Queue Precedence .....	148



# *Preface*

## **The Open Group**

The Open Group is an international open systems organisation that is leading the way in creating the infrastructure needed for the development of network-centric computing and the information superhighway. Formed in 1996 by the merger of the X/Open Company and the Open Software Foundation, The Open Group is supported by most of the world's largest user organisations, information systems vendors and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to assist user organisations, vendors and suppliers in the development and implementation of products supporting the adoption and proliferation of open systems.

With more than 300 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritising and communicating customer requirements to vendors
- conducting research and development with industry, academia and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the X/Open brand that designates vendor products which conform to X/Open Product Standards
- promoting the benefits of open systems to customers, vendors and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trade mark on behalf of the industry.

## **The X/Open Process**

This description is used to cover the whole Process developed and evolved by X/Open. It includes the identification of requirements for open systems, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The X/Open brand logo is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the X/Open brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

### Open Group Publications

The Open Group publishes a wide range of technical literature, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys and business titles.

There are several types of specification:

- *CAE Specifications*

CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our product standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. In addition, they can demonstrate product compliance through the X/Open brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organisations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif and CDE.

Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

This includes product documentation — programmer's guides, user manuals, and so on — relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

These provide information that is useful in the evaluation, procurement, development or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Programme. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

- *Snapshots*

These provide a mechanism to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of a technical activity.

### **Versions and Issues of Specifications**

As with all *live* documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

### **Corrigenda**

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at <http://www.opengroup.org/public/pubs>.

### **Ordering Information**

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at <http://www.opengroup.org/public/pubs>.

## This Document

This document is a CAE Specification. It specifies a STREAMS kernel-level instantiation of the ISO Data Link Service Definition ISO/IEC 8886 and Logical Link Control ISO/IEC 8802-2 (LLC). Where the two standards do not conform, ISO/IEC 8886 prevails.

The Data Link Provider Interface (DLPI) enables a data link service user to access and use any of a variety of conforming data link service providers without special knowledge of the provider's protocol.

Specifically, the interface is intended to support X.25 LAPB, BX.25 level 2, SDL C, ISDN LAPD, Ethernet, CSMA/CD, FDDI, token ring, token bus, Bisync, Frame Relay, ATM, Fiber Channel and HIPPI. Readers should note that the list of data links supported by the interface may be augmented and is published on the World-Wide Web in the DLPI Corrigendum **U019** at <http://www.opengroup.org> (under *Sales and Ordering*). This Corrigendum will contain relevant additional information such as recognised implementors' agreements/guidelines for common data link providers. A printed copy of this Web page may also be obtained from The Open Group on request (address on inside front page of this specification).

Among the expected data link service users are implementations of the OSI network layer and SNA path control.

The interface specifies access to data link service providers, and does not define a specific protocol implementation. Thus, issues of network management, protocol performance, and performance analysis tools are beyond the scope of this document and should be addressed by specific implementations of a data link provider. However, accompanying each provider implementation should be information that describes the protocol-specific behavior of that provider. Currently, there are plans to come up with a set of implementors' agreements/guidelines for common data link providers. These agreements will address issues such as DLSAP address space, subsequent addresses, ppa access and control, QOS, supported services etc.

## Intended Audience

This specification assumes the reader is familiar with OSI Reference Model terminology, OSI Data Link Services, and STREAMS.

## Document Structure

- **Chapter 1** presents background on the structure of the data link layer of the OSI Reference Model, and explains the intended architecture in the STREAMS environment. Data link addressing concepts are also presented.
- **Chapter 2** presents an overview of the services provided by DLPI.
- **Chapter 3** describes the detailed syntax and semantics of each DLPI primitive that crosses the data link interface.
- **Chapter 4** describes the quality-of-service parameters supported by DLPI and the rules for negotiating/selecting the values of those parameters.
- **Appendix A** lists the optional primitives to perform certain essential management functions.
- **Appendix B** describes the allowable sequence of DLPI primitives that may be issued across the interface.
- **Appendix C** presents a summary of the precedence of DLPI primitives as they are queued by the DLS provider and/or DLS user.

- **Appendix D** summarizes guidelines a DLS user implementation must follow to be fully protocol-independent.
- **Appendix E** identifies the information which should be documented for each DLS provider implementation.
- **Appendix F** presents the header file containing DLPI structure and constant definitions needed by a DLS user or provider implemented to use the interface.

A Glossary and Index are also provided.

### Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - variable names, for example, substitutable argument prototypes and environment variables
  - C-language functions; these are shown as follows: *name()*
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a C-language header file.
- Syntax, code examples and user input in interactive examples are shown in fixed width font. Brackets shown in this font, [ ], are part of the syntax and do *not* indicate optional items. In syntax, the | symbol is used to separate alternatives, and ellipses (. . .) are used to show that additional arguments are optional.

## *Trade Marks*

Motif<sup>®</sup>, OSF/1<sup>®</sup> and UNIX<sup>®</sup> are registered trade marks and the “X Device”<sup>™</sup> and The Open Group<sup>™</sup> are trade marks of The Open Group.

# *Acknowledgements*

The base document from which this specification is derived was published as Copyright 1991 UNIX International, Inc.

The following copyright notice was attached to the document:

Permission to use, copy, modify, and distribute this documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name UNIX International not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. UNIX International makes no representations about the suitability of this documentation for any purpose. It is provided "as is" without express or implied warranty.

## *Referenced Documents*

The following documents are referenced in this specification:

### ISO/IEC 8886

ISO/IEC 8886: 1996, Information Technology — Open Systems Interconnection — Data Link Service Definition.

### ISO/IEC 8802-2

ISO/IEC 8802-2: 1994, Information Technology — Telecommunications and Information Exchange between Systems — Local and Metropolitan area Networks — Specific Requirements — Part 2: Logical Link Control.

### STREAMS

UNIX Press (A Prentice Hall Title) book "STREAMS Modules and Drivers", published 1992, ISBN 0-13-066879-6.

### X.200

CCITT Recommendation X.200, "Reference Model of Open Systems Interconnection for CCITT Applications", 1984.

### XNS, Issue 5

CAE Specification, February 1997, Networking Services, Issue 5 (ISBN: 1-85912-165-9, C523).

## 1.1 Overview

This document specifies a STREAMS kernel-level instantiation of the ISO Data Link Service Definition ISO/IEC 8886 (see referenced documents) and Logical Link Control ISO/IEC 8802-2 (LLC) (see referenced documents). Where the two standards do not conform, ISO/IEC 8886 prevails.

The Data Link Provider Interface (DLPI) enables a data link service user to access and use any of a variety of conforming data link service providers without special knowledge of the provider's protocol.

Specifically, the interface is intended to support X.25 LAPB, BX.25 level 2, SDLC, ISDN LAPD, Ethernet, CSMA/CD, FDDI, token ring, token bus, Bisync, Frame Relay, ATM, Fiber Channel and HIPPI. Among the expected data link service users are implementations of the OSI network layer and SNA path control. The interface specifies access to data link service providers, and does not define a specific protocol implementation. Thus, issues of network management, protocol performance, and performance analysis tools are beyond the scope of this document and should be addressed by specific implementations of a data link provider. However, accompanying each provider implementation should be information that describes the protocol-specific behavior of that provider. Currently, there are plans to come up with a set of implementors' agreements/guidelines for common data link providers. These agreements will address issues such as DLSAP address space, subsequent addresses, PPA access and control, QOS, supported services etc.

Readers should note that the list of data links supported by the interface may be augmented and is published on the World-Wide Web in the DLPI Corrigendum **U019** at [f5http://www.xopen.org](http://www.xopen.org) (under *Sales and Ordering*). This Corrigendum will contain relevant additional information such as recognised implementors' agreements/guidelines for common data link providers. A printed copy of this Web page may also be obtained from The Open Group on request (address on inside front page of this specification).

This specification assumes the reader is familiar with OSI Reference Model (see referenced documents) terminology, OSI Data Link Services, and STREAMS. Although there are no formal standards for a STREAMS environment, extensive descriptions of STREAMS and STREAMS programming can be found in the referenced document **STREAMS**.

## 1.2 Model of 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.

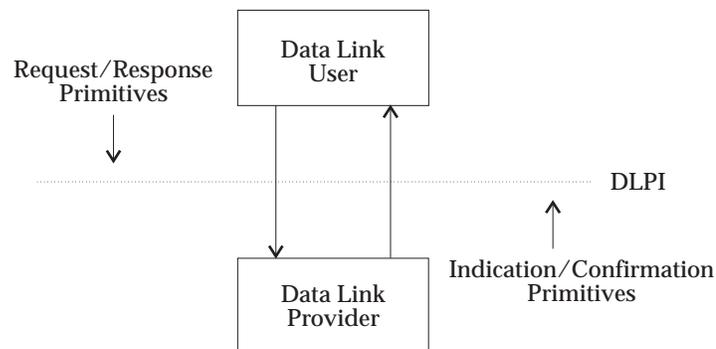
The model of the data link layer is presented here to describe concepts that are used throughout the specification of DLPI. It is described in terms of an interface architecture, as well as addressing concepts needed to identify different components of that architecture. The description of the model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

### 1.2.1 Model of 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 depicts the abstract view of DLPI.



**Figure 1-1** Abstract View of DLPI

The data link interface is the boundary between the network and 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 which 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, service interfaces that map to the Data Link and Transport Layers of the OSI Reference Model have been developed. 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()* and *getmsg()* system calls.

Referring to the abstract view of DLPI (see Figure 1-1), the DLS provider is configured as a STREAMS driver, and the DLS user accesses the provider using *open()* 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 the messages presented later in this specification.

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 independently 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.

## 1.3 Modes of Communication

The data link provider interface supports three modes of communication:

- connection
- connectionless
- acknowledged connectionless.

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, however, 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 acknowledgement 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.

The acknowledged connectionless mode provides the means by which a data link user can send data and request the return of data at the same time. Although the exchange service is connectionless, in-sequence delivery is guaranteed for data sent by the initiating station. The data unit transfer is point-to-point.

### 1.3.1 Connection-mode Service

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

- local management
- connection establishment
- data transfer
- connection release.

#### 1.3.1.1 Local Management

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

#### 1.3.1.2 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 (see Section 1.4 on page 6).

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 may 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.

#### 1.3.1.3 *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.

#### 1.3.1.4 *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.

### 1.3.2 **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 (see Section 1.4 on page 6).

Connectionless data transfer does not guarantee that data units will be delivered to the destination user in the order in which they were sent. Furthermore, it does not guarantee that a given data unit will reach the destination DLS user, although a given DLS provider may provide assurance that data will not be lost.

### 1.3.3 **Acknowledged Connectionless-mode Service**

The acknowledged connectionless mode service also 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, the acknowledged connectionless data transfer phase is immediately entered.

Acknowledged connectionless data transfer guarantees that data units will be delivered to the destination user in the order in which they were sent. A data link user entity can send a data unit to the destination DLS User, request a previously prepared data unit from the destination DLS User, or exchange data units.

## 1.4 DLPI Addressing

Each user of DLPI must establish an identity to communicate with other data link users. This identity consists of two pieces. Firstly, 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. Secondly, 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.

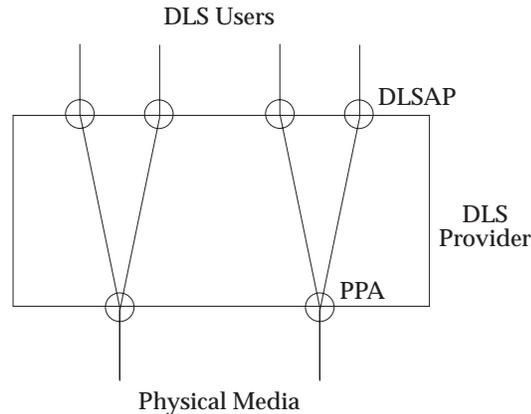


Figure 1-2 Data Link Addressing Components

### 1.4.1 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. One possible implementation of a style 1 driver would reserve a major device for each PPA the data link driver would support. This would allow the STREAMS clone open feature to be used for each PPA configured. 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()* 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, and should be described in the provider-specific addendum documentation.

DLPI provides a mechanism to get or modify the physical address. The primitives to handle these functions are described in Appendix A. The physical address value can be modified in a post-attached state. This would modify 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, 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 by any means that it chooses the isolation of SAP and physical address space effects on a per-stream basis.

### 1.4.2 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 DLSAP address, or to determine the DLSAP associated with a stream by retrieving the bound DLSAP address. This 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, and should be described in the provider-specific addendum documentation. 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.

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 allows 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 *ether\_type* value of *IP*, and a subsequent bind could be issued with *ether\_type* value of *ARP*. The Provider may now multiplex off of 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* specifies a DLSAP that will be used in addition to the DLSAP bound using a DL\_BIND\_REQ. This allows additional information to be specified, which will be used in a header or for demultiplexing. An example of this would be to use hierarchical bind to specify the OUI (Organizationally Unique Identifier) 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.

## 1.5 Connection Management Stream

The earlier description of the connection-mode service assumed that a DLS user bound a DLSAP to the stream it would use to receive connect requests. In some instances, however, it is expected that a given service may be accessed through any one of several DLSAPs. To handle this scenario, a separate stream would be required for each possible destination DLSAP, regardless of whether any DLS user actually requested a connection to that DLSAP. Obvious resource problems can result in this scenario.

To obviate the need for tying up system resources for all possible destination DLSAPs, a *connection management stream* utility is defined in DLPI. A connection management stream is one that receives any connect requests that are not destined for currently bound DLSAPs capable of receiving connect indications. With this mechanism, a special listener can handle incoming connect requests intended for a set of DLSAPs, by opening a connection management stream to the DLS provider which will retrieve all connect requests arriving through a particular PPA. In the model, therefore, there may be a connection management stream per PPA.

## *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.

The data link provider interface supports three modes of service: connection, connectionless and acknowledged connectionless. 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. The acknowledged connectionless mode is message-oriented and guarantees that data units will be delivered to the destination user in the order in which they were sent. This specification also defines a set of local management functions that apply to all modes of service.

The XID and TEST services that are supported by DLPI are listed below. 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.

The services are tabulated below and described more fully in the remainder of this section.

Phase	Service	Primitives
Local Management	Information Reporting	DL_INFO_REQ, DL_INFO_ACK, DL_ERROR_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
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
	QOS Management	DL_UDQOS_REQ, DL_OK_ACK, DL_ERROR_ACK
	Error Reporting	DL_UDERROR_IND
XID and TEST services	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
Acknowledged Connectionless-mode Data Transfer	Data Transfer	DL_DATA_ACK_REQ, DL_DATA_ACK_IND, DL_DATA_ACK_STATUS_IND, DL_REPLY_REQ, DL_REPLY_IND, DL_REPLY_STATUS_IND, DL_REPLY_UPDATE_REQ, DL_REPLY_UPDATE_STATUS_IND
	QOS Management	DL_UDQOS_REQ, DL_OK_ACK, DL_ERROR_ACK
	Error Reporting	DL_UDERROR_IND

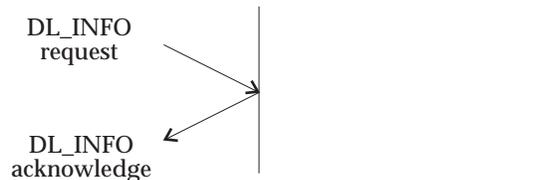
**Table 2-1** Cross-Reference of DLS Services and Primitives

## 2.1 Management Services

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

### 2.1.1 Information Reporting Service

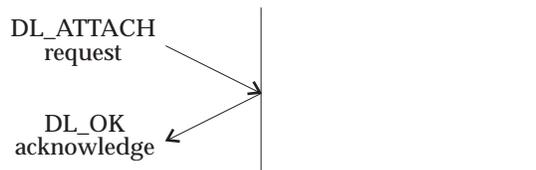
This service provides information about the DLPI stream to the DLS user. The message DL\_INFO\_REQ requests the DLS provider to return operating information about the stream. The DLS provider returns the information in a DL\_INFO\_ACK message.



**Figure 2-1** Information Reporting

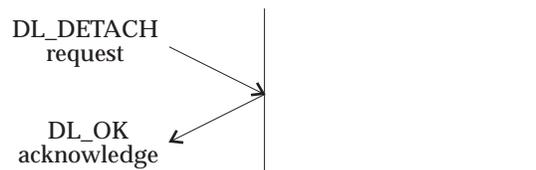
### 2.1.2 Attach Service

The attach service assigns a physical point of attachment ( PPA ) to a stream. This service is required for style 2 DLS providers (see Section 1.4.1 on page 6) to specify the physical medium over which communication will occur. The DLS provider indicates success with a DL\_OK\_ACK; failure with a DL\_ERROR\_ACK. The normal message sequence is illustrated in the following figure.



**Figure 2-2** 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 the following figure.



**Figure 2-3** Detaching a Stream from a Physical Line

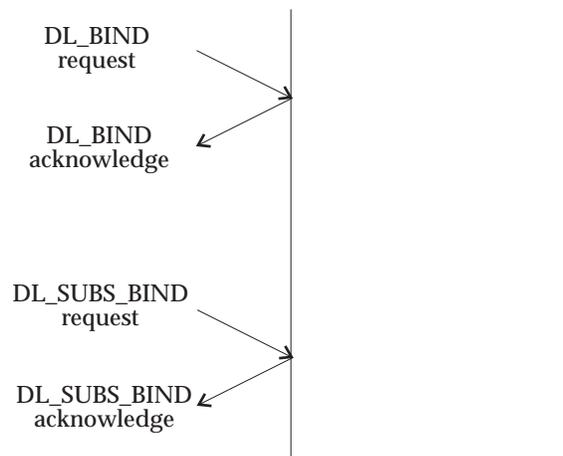
### 2.1.3 Bind Service

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 and acknowledged connectionless data transfer and connection establishment requests. Protocol-specific actions taken during activation should be described in DLS provider-specific addenda.

The DLS provider indicates success with a DL\_BIND\_ACK; failure with a DL\_ERROR\_ACK. Certain DLS providers require the capability of binding on multiple DLSAP addresses. DL\_SUBS\_BIND\_REQ provides that added capability. The DLS provider indicates success with a DL\_SUBS\_BIND\_ACK; failure with a DL\_ERROR\_ACK.

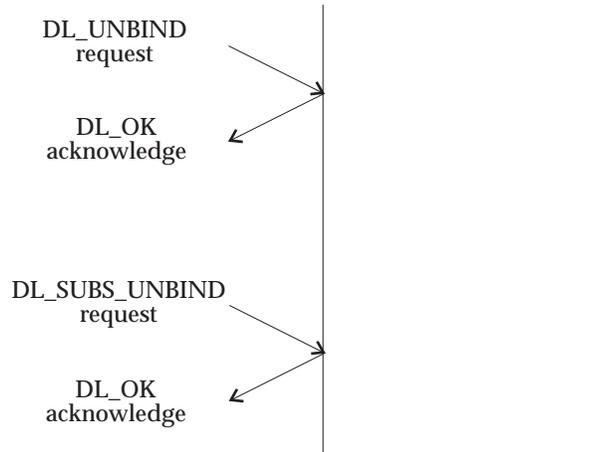
The normal flow of messages is illustrated in the following figure.



**Figure 2-4** Binding a Stream to a DLSAP

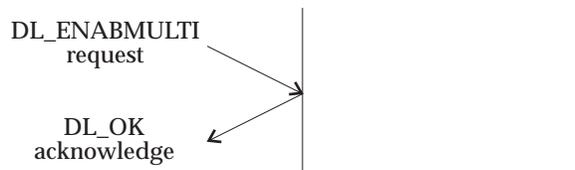
DL\_UNBIND\_REQ requests the DLS provider to unbind all DLSAP(s) from a stream. The DL\_UNBIND\_REQ also unbinds all the subsequently bound DLSAPs that have not been unbound. The DLS provider indicates success with a DL\_OK\_ACK; failure with a DL\_ERROR\_ACK.

DL\_SUBS\_UNBIND\_REQ requests the DLS Provider to unbind the subsequently bound DLSAP. The DLS Provider indicates success with a DL\_OK\_ACK; failure with a DL\_ERROR\_ACK.



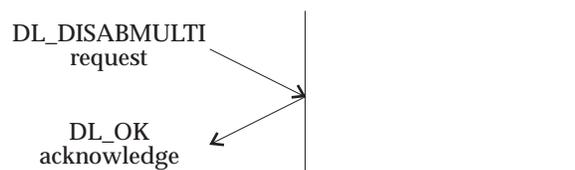
**Figure 2-5** Unbinding a Stream from a DLSAP

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; failure with a DL\_ERROR\_ACK.



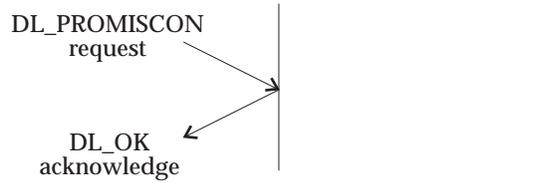
**Figure 2-6** Enabling a specific multicast address on a Stream

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; failure with a DL\_ERROR\_ACK.



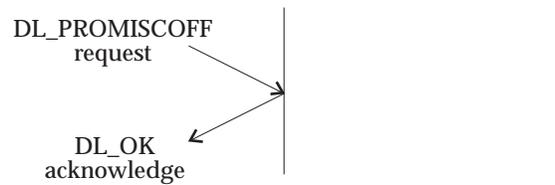
**Figure 2-7** Disabling a specific multicast address on a Stream

DL\_PROMISCON\_REQ requests the DLS Provider to enable promiscuous mode on a per Stream basis, either at the physical level or at the SAP level. The Provider indicates success with a DL\_OK\_ACK; failure with a DL\_ERROR\_ACK.



**Figure 2-8** 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 or at the SAP level. The Provider indicates success with a DL\_OK\_ACK; failure with a DL\_ERROR\_ACK.



**Figure 2-9** Disabling promiscuous mode on a Stream

## 2.2 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.

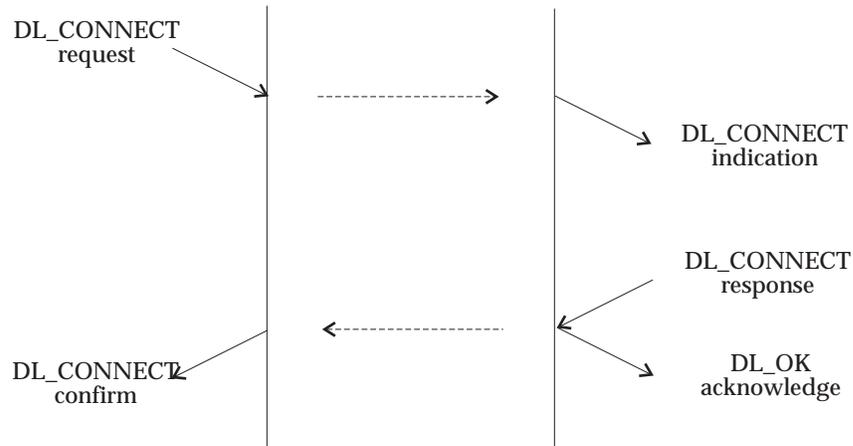
### 2.2.1 Connection Establishment Service

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.

#### 2.2.1.1 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 informing the calling DLS user that the connection has been established.

The normal sequence of messages is illustrated in the following figure.



**Figure 2-10** 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 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. The normal sequence of messages for obtaining a token is illustrated in the following figure.

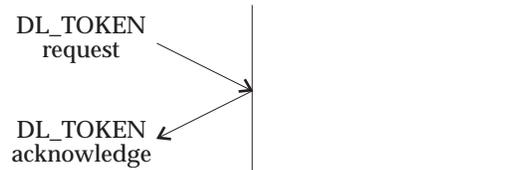


Figure 2-11 Token Retrieval

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.

2.2.1.2 Connection Establishment Rejections

In certain situations, the connection establishment request cannot be completed. The following paragraphs describe the occasions under which DL\_DISCONNECT\_REQ and DL\_DISCONNECT\_IND primitives will flow during connection establishment, causing the connect request to be aborted.

The following figure illustrates the situation where the called DLS user chooses to reject the connect request by issuing DL\_DISCONNECT\_REQ instead of DL\_CONNECT\_RES.

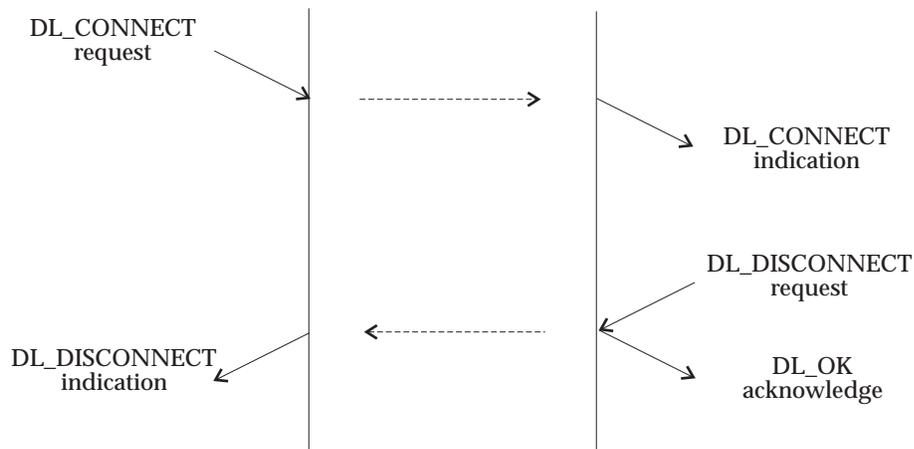
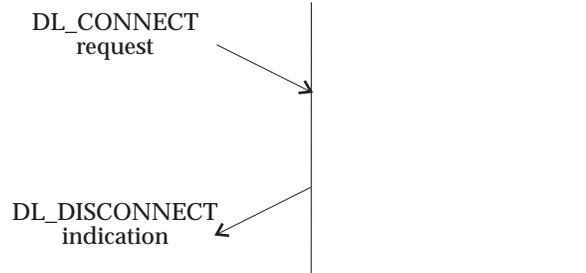


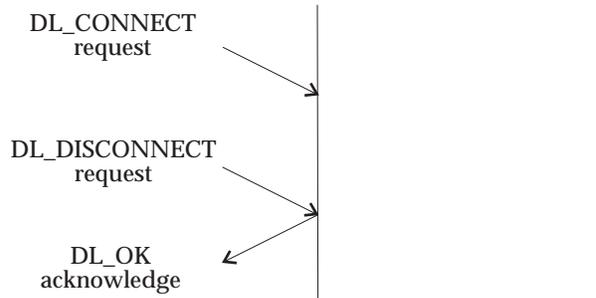
Figure 2-12 Called DLS User Rejection of Connection Establishment Attempt

The following figure illustrates the situation where the DLS provider rejects a connect request for lack of resources or other reason. The DLS provider sends DL\_DISCONNECT\_IND in response to DL\_CONNECT\_REQ.

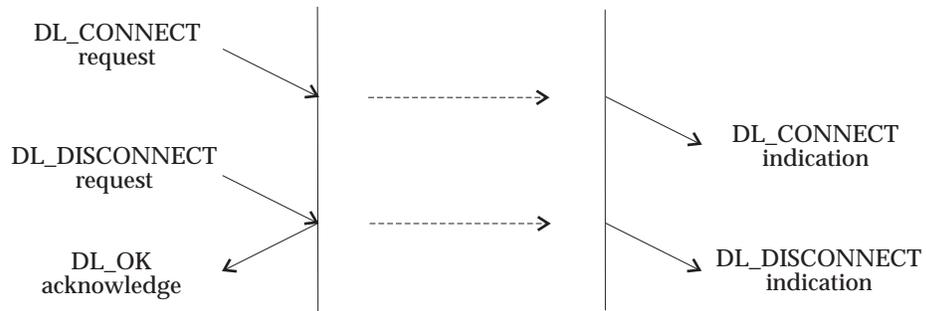


**Figure 2-13** DLS Provider Rejection of a Connection Establishment Attempt

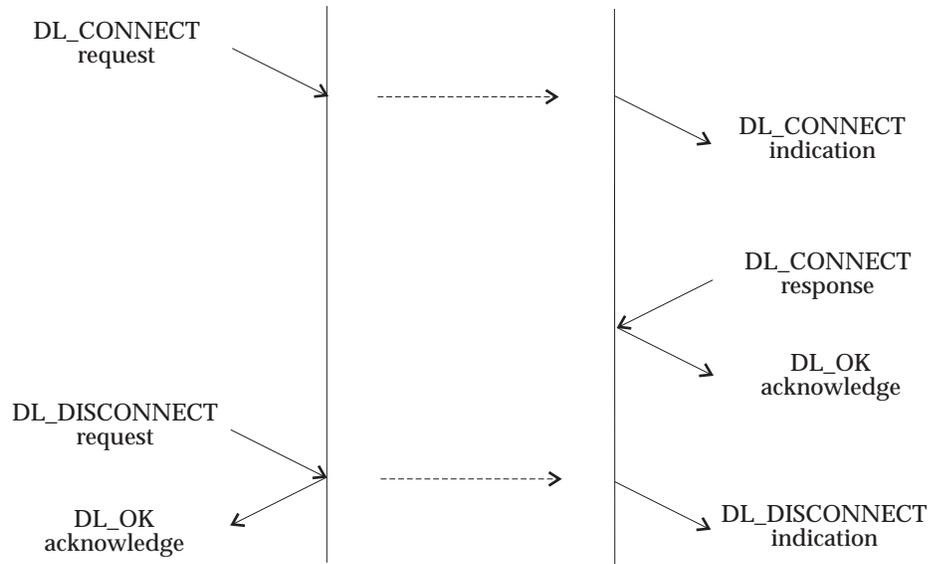
The following figures illustrate the situation where the calling DLS user chooses to abort a previous connection attempt. The DLS user issues DL\_DISCONNECT\_REQ at some point following a DL\_CONNECT\_REQ. The resulting sequence of primitives depends on the relative timing of the primitives involved, as defined in the following time sequence diagrams.



**Figure 2-14** Both Primitives Destroyed by Provider



**Figure 2-15** DL\_DISCONNECT Indication Arrives before DL\_CONNECT Response is Sent



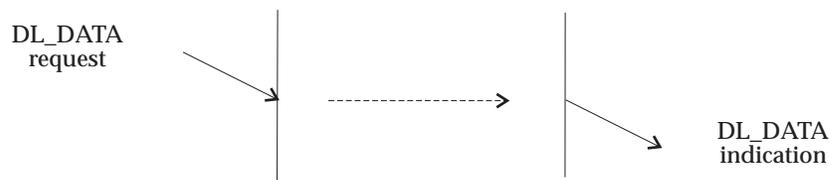
**Figure 2-16** DL\_DISCONNECT Indication Arrives after DL\_CONNECT Response is Sent

**2.2.2 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 (DLSUs). The DLS provider preserves both the sequence and boundaries of DLSUs 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 DLSU from the local DLS user to the DLS provider. Similarly, each DL\_DATA\_IND primitive conveys a DLSU from the DLS provider to the remote DLS user. The normal flow of messages is illustrated in the figure below.



**Figure 2-17** Normal Data Transfer

### 2.2.3 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 the following figure.



Figure 2-18 DLS User-Invoked Connection Release

The next figure illustrates that when two DLS users independently invoke the connection release service, neither receives a DL\_DISCONNECT\_IND.

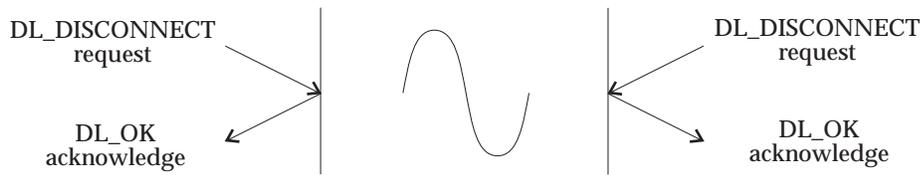


Figure 2-19 Simultaneous DLS User Invoked Connection Release

The next figure illustrates that when the DLS provider initiates the connection release service, each DLS user receives a DL\_DISCONNECT\_IND.

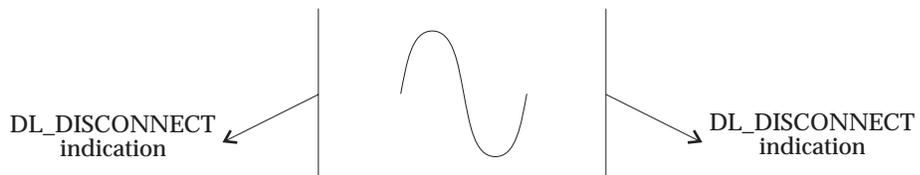
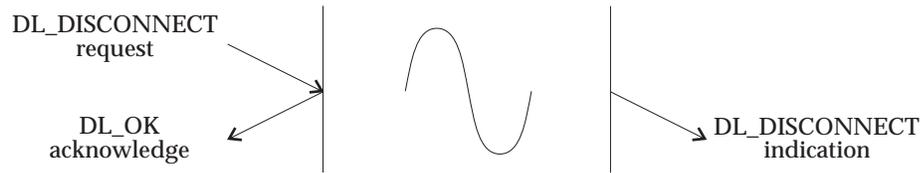


Figure 2-20 DLS Provider Invoked Connection Release

The next figure 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 2-21** Simultaneous DLS User & DLS Provider Invoked Connection Release Service

#### 2.2.4 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.

Invocation of the reset service will unblock the flow of DLSDUs if the data link connection is 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 an asynchronization mark in the stream of DLSDUs that are transmitted by the issuing DLS user; the DL\_RESET\_IND acts as an asynchronization mark in the stream of DLSDUs that are received by the peer DLS user. Similarly, the DL\_RESET\_RES acts as an asynchronization 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 that:

- 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. The following figure illustrates the message flow for a reset invoked by one DLS user.

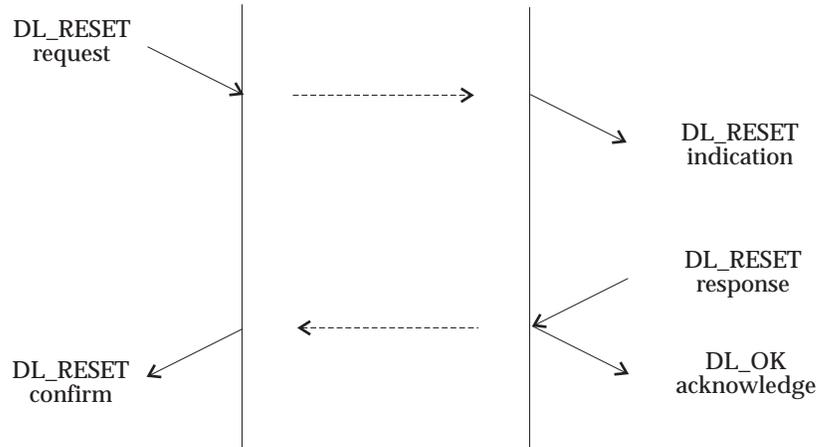


Figure 2-22 DLS User-Invoked Connection Reset

The following figure illustrates the message flow for a reset invoked by both DLS users simultaneously.

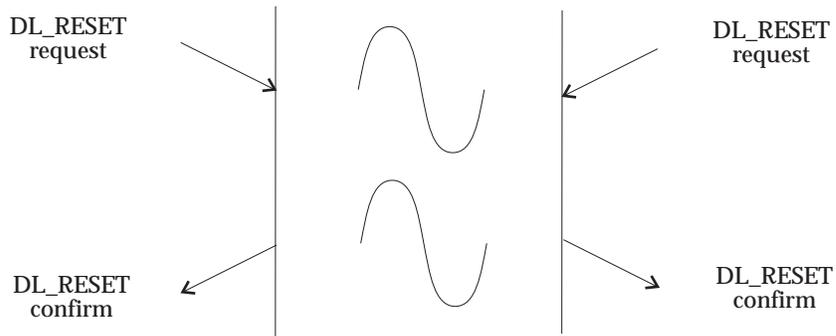
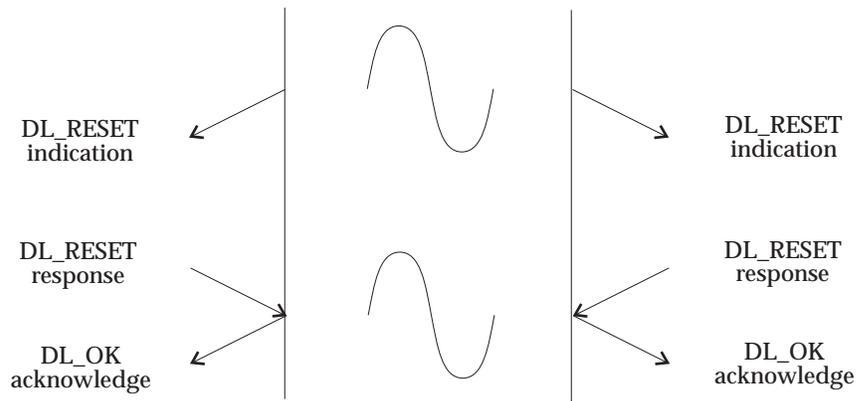


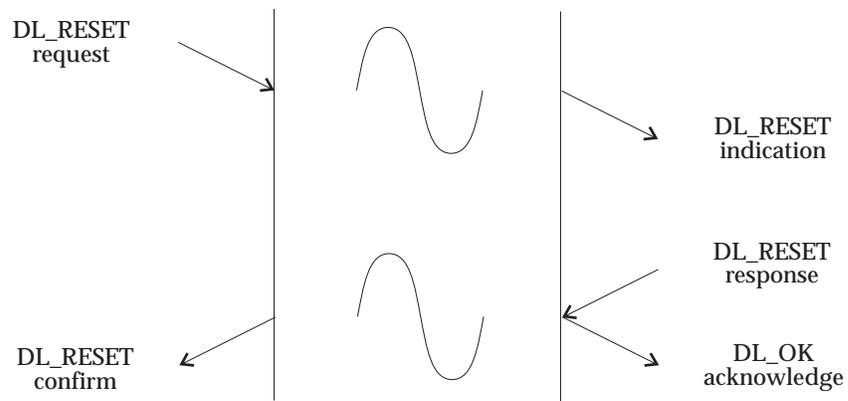
Figure 2-23 Simultaneous DLS User-Invoked Connection Reset

The following figure illustrates the message flow for a reset invoked by the DLS provider.



**Figure 2-24** DLS Provider-Invoked Connection Reset

The following figure illustrates the message flow for a reset invoked simultaneously by one DLS user and the DLS provider.



**Figure 2-25** Simultaneous DLS User & DLS Provider Invoked Connection Reset

## 2.3 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 receive connectionless data units.

### 2.3.1 Connectionless Data Transfer Service

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 the figure below.

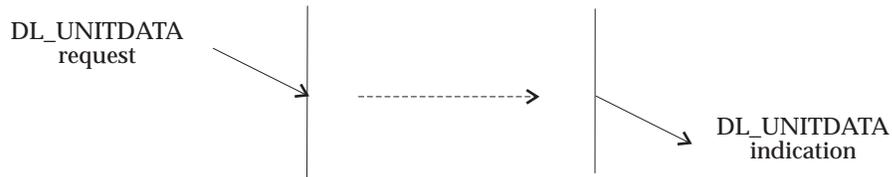


Figure 2-26 Connectionless Data Transfer

### 2.3.2 QOS Management Service

The QOS (Quality of Service) management service enables a DLS user to specify the quality of service it can expect for each invocation of the connectionless data transfer service. The DL\_UDQOS\_REQ directs the DLS provider to set the QOS parameters to the specified values. The normal flow of messages is illustrated in the figure below.

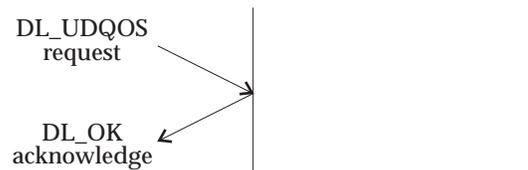


Figure 2-27 QOS Service

### 2.3.3 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.

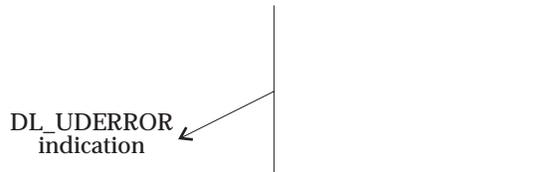


Figure 2-28 Error Reporting Service

### 2.3.4 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 XID 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 acknowledgement 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.

The normal flow of messages is illustrated in the figures below.

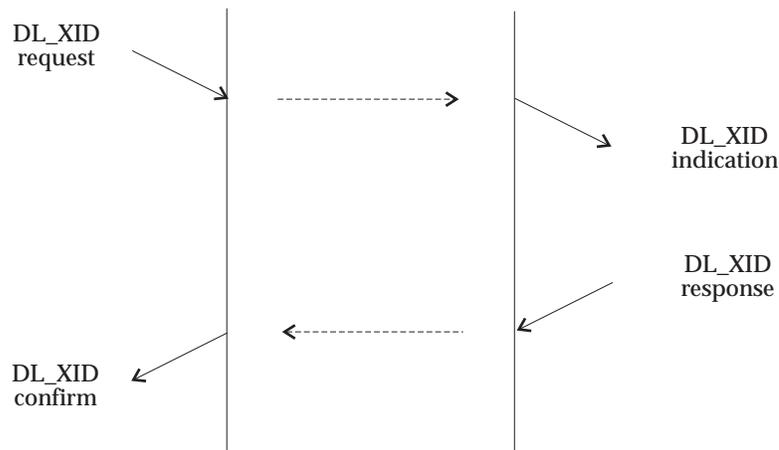


Figure 2-29 XID Service

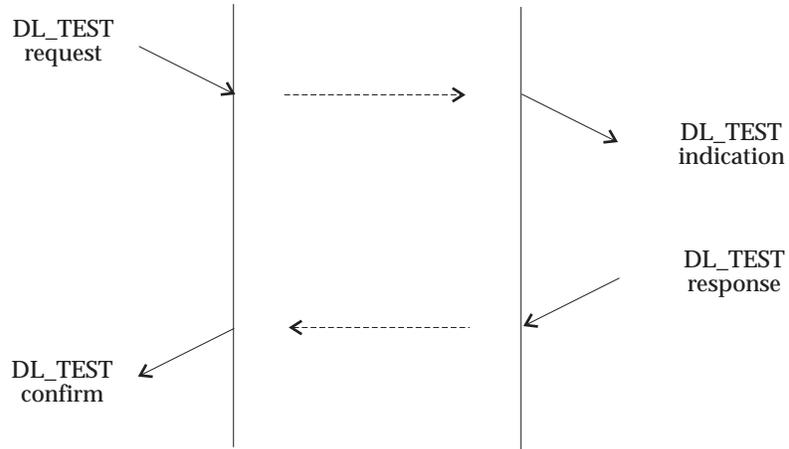


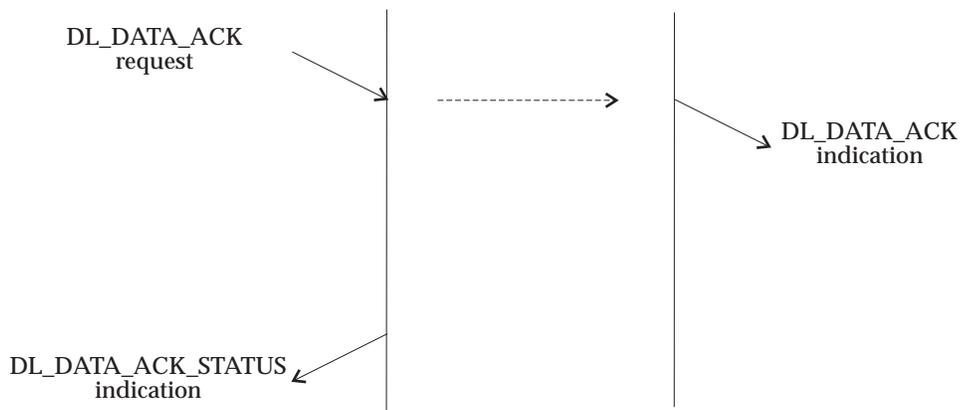
Figure 2-30 TEST Service

## 2.4 Acknowledged Connectionless-mode Services

The acknowledged connectionless-mode services are designed for general use for the reliable transfer of informations between peer DLS Users. These services are intended for applications that require acknowledgement of cross-LAN data unit transfer, but wish to avoid the complexity that is viewed as being associated with the connection-mode services. Although the exchange service is connectionless, insequence delivery is guaranteed for data sent by the initiating station.

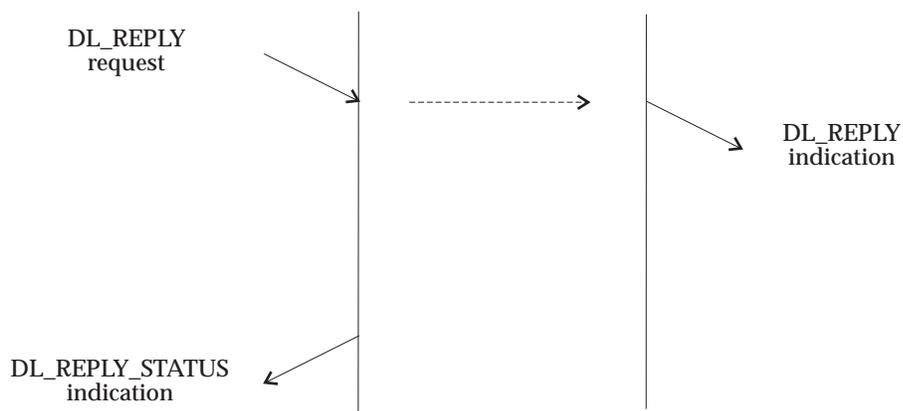
### 2.4.1 Acknowledged Connectionless-mode Data Transfer Services

The acknowledged connectionless-mode data transfer services provide the means by which the DLS Users can exchange DLSDUs which are acknowledged at the LLC sublayer, without the establishment of a Data Link connection. The services provide a means by which a local DLS User can send a data unit to the peer DLS User, request a previously prepared data unit, or exchange data units with the peer DLS User.



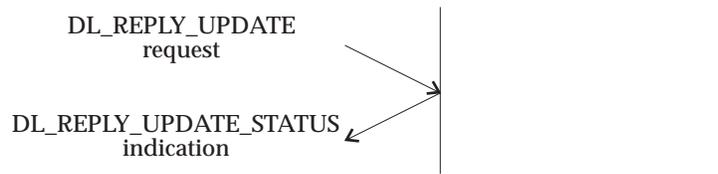
**Figure 2-31** Acknowledged Connectionless-Mode Data Unit Transmission service

The next figure illustrates the acknowledged connectionless-mode data unit exchange service.



**Figure 2-32** Acknowledged Connectionless-Mode Data Unit Exchange service

The next figure illustrates the Reply Data Unit Preparation service.



**Figure 2-33** Acknowledged Connectionless-Mode Reply Data Unit Preparation Service

#### 2.4.2 QOS Management Service

The Quality of Service (QOS) management service enables a DLS User to specify the quality of service it can expect for each invocation of the acknowledged connectionless data transfer service. The DL\_UDQOS\_REQ directs the DLS provider to set the QOS parameters to the specified values. The normal flow of messages is illustrated in Section 2.3.2 on page 23).

#### 2.4.3 Error Reporting Service

The acknowledged connectionless mode error reporting service is the same as the unacknowledged connectionless-mode error reporting service. For the message flow, refer to Section 2.3.3 on page 24.

## 2.5 Example

To bring together all the the items described in this chapter, the following example illustrates the primitives that flow during a complete, connection-mode sequence between stream open and stream close.

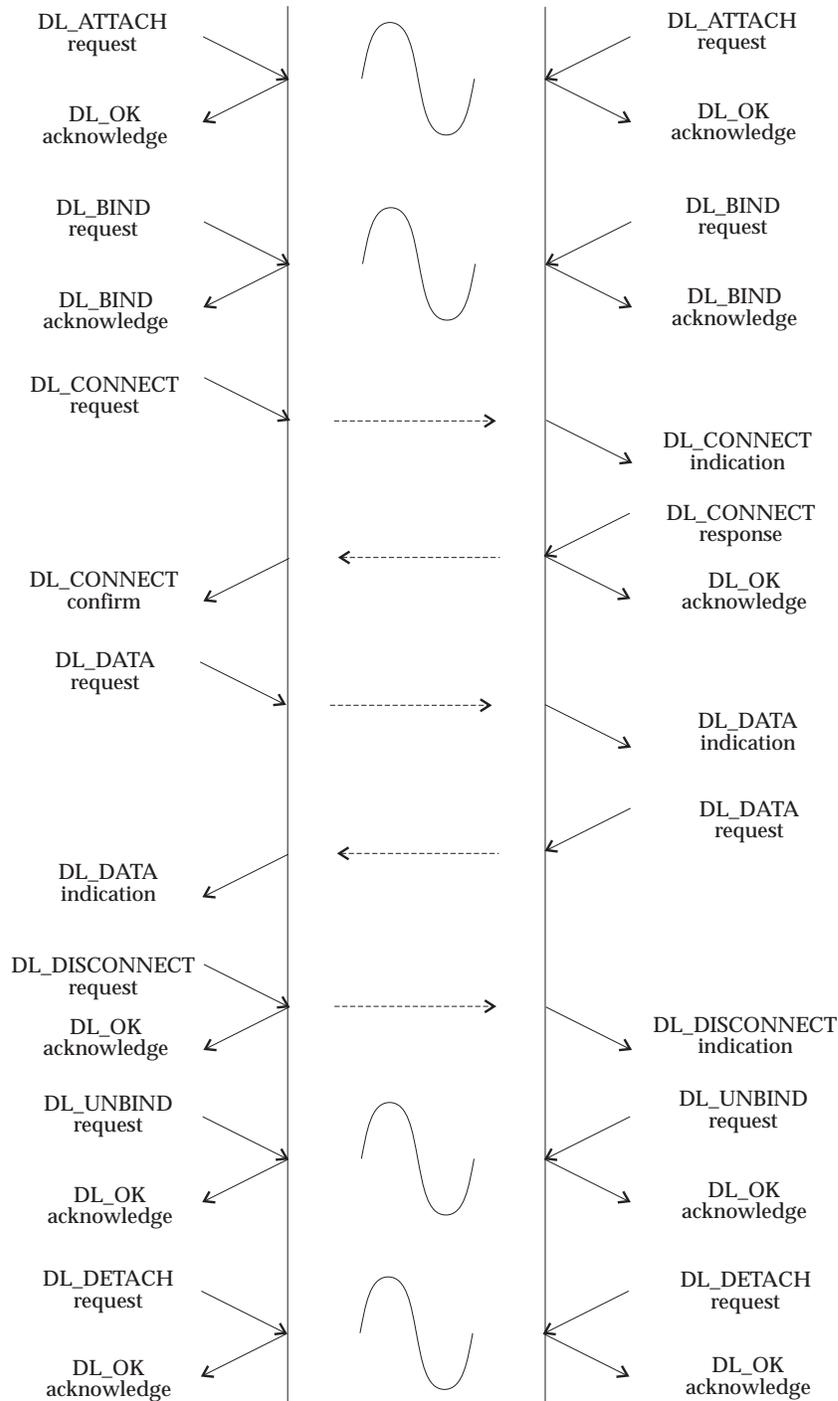


Figure 2-34 A Connection-mode Example

Discard this page.

Discard this page too.

## 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()*. 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 information contained in the M\_PROTO or M\_PCPROTO message blocks must begin on a byte boundary that is appropriate for structure alignment (for example, word-aligned on the AT&T 3B2 Computer). STREAMS will allocate buffers that begin on such a boundary. However, these message blocks may contain information whose representation is described by a length and an offset within the block. An example is the DLSAP address (*dl\_addr\_length* and *dl\_addr\_offset*) in the DL\_BIND\_ACK primitive. The offset of such information within the message block is not guaranteed to be properly aligned for casting the appropriate data type (such as an *int* or a *struct*). Appendix B defines the sequence in which DLPI primitives can be passed between DLS user and DLS provider, and Appendix C summarizes the precedence rules associated with each primitive for ordering the primitives on the DLS provider and DLS user queues.

The following sections describe the format of the primitives that support the services described in the previous chapter. The primitives are grouped into four general categories for presentation:

- Local Management Service Primitives
- Connection-mode Service Primitives
- Connectionless-mode Service Primitives
- Acknowledged Connectionless-mode Service Primitives

Three **types** are used to build the DLPI primitives. The normative definitions of **t\_scalar\_t** and **t\_uscalar\_t** are to be found in the Networking Services Specification (see the referenced XNS specification), but are repeated here for informational purposes. **uint16\_t** is one of the fixed width types defined in `<sys/inttypes.h>`.

**t\_scalar\_t** and **t\_uscalar\_t** are, respectively, a signed and an unsigned opaque integral type of equal length of at least 32 bits<sup>1</sup>.

---

1. To forestall portability problems, it is recommended that applications should not use values larger than  $2^{32} - 1$ .

### 3.1 Local Management Service Primitives

This section describes the local management service primitives that are common to the *connection*, *connectionless* and *acknowledged connectionless* service modes. These primitives support the Information Reporting, Attach, Bind, Enabling/Disabling of multicast addresses, and Turning On/Off the promiscuous mode. Once a stream has been opened by a DLS user, these primitives initialize the stream, preparing it for use.

#### 3.1.1 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 either of the following methods:

- pre-initialized by some network management mechanism before the DL\_BIND\_REQ is received
- 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 one of 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
- 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.

**3.1.2 List of Local Management Service Primitives**

The local management service primitives are listed below and are defined in the following man-pages:

- DL\_INFO\_REQ
- DL\_INFO\_ACK
- DL\_ATTACH\_REQ
- DL\_DETACH\_REQ
- DL\_BIND\_REQ
- DL\_BIND\_ACK
- DL\_UNBIND\_REQ
- DL\_SUBS\_BIND\_REQ
- DL\_SUBS\_BIND\_ACK
- DL\_SUBS\_UNBIND\_REQ
- DL\_ENABMULTI\_REQ
- DL\_DISABMULTI\_REQ
- DL\_PROMISCON\_REQ
- DL\_PROMISCOFF\_REQ
- DL\_OK\_ACK
- DL\_ERROR\_ACK

**NAME**

DL\_INFO\_REQ (*dl\_info\_req\_t*) — request 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.

**SYNOPSIS**

The message consists of one M\_PCPROTO message block, which contains the following structure:

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

**PARAMETERS**

*dl\_primitive*  
conveys DL\_INFO\_REQ.

**STATE**

The message is valid in any state in which a local acknowledgement is not pending, as described in Appendix B on page 131.

**NEW STATE**

The resulting state is unchanged.

**RESPONSE**

The DLS provider responds to the information request with a DL\_INFO\_ACK.

**NAME**

DL\_INFO\_ACK (*dl\_info\_ack\_t*) — this message is sent in response to DL\_INFO\_REQ; it conveys information about the DLPI stream to the DLS user.

**SYNOPSIS**

The message consists of one M\_PCPROTO message block, which contains the following structure.

```
typedef struct {
    t_uscalar_t dl_primitive;
    t_uscalar_t dl_max_sdu;
    t_uscalar_t dl_min_sdu;
    t_uscalar_t dl_addr_length;
    t_uscalar_t dl_mac_type;
    t_uscalar_t dl_reserved;
    t_uscalar_t dl_current_state;
    t_uscalar_t dl_sap_length;
    t_uscalar_t dl_service_mode;
    t_uscalar_t dl_qos_length;
    t_uscalar_t dl_qos_offset;
    t_uscalar_t dl_qos_range_length;
    t_uscalar_t dl_qos_range_offset;
    t_uscalar_t dl_provider_style;
    t_uscalar_t dl_addr_offset;
    t_uscalar_t dl_version;
    t_uscalar_t dl_brdcst_addr_length;
    t_uscalar_t dl_brdcst_addr_offset;
    t_uscalar_t dl_growth;
} dl_info_ack_t;
```

**PARAMETERS***dl\_primitive*

conveys DL\_INFO\_ACK.

*dl\_max\_sdu*

conveys the maximum number of bytes that may be transmitted in a DLSDU. This value must be a positive integer that is greater than or equal to the value of *dl\_min\_sdu*.

*dl\_min\_sdu*

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

*dl\_addr\_length*

conveys the length, in bytes, of the provider's DLSAP address. In the case of ahierarchical subsequent bind, the length returned is the total length, that is, Physical address + SAP +subsequent address length.

*dl\_mac\_type*

conveys the type of medium supported by this DLPI stream<sup>2</sup>. Possible values include:

2. Readers should note that the list of data links supported by the interface may be augmented and is published on the World-Wide Web in **U019** at <http://www.xopen.org> (under *Sales and Ordering*). This Corrigendum will contain relevant additional information such as recognised implementors' agreements/guidelines for common data link providers.

- DL\_CSMACD  
The medium is Carrier Sense Multiple Access with Collision Detection (ISO 8802/3).
- DL\_TPB  
The medium is Token-Passing Bus (ISO 8802/4).
- DL\_TPR  
The medium is Token-Passing Ring (ISO 8802/5).
- DL\_METRO  
The medium is Metro Net (ISO 8802/6).
- DL\_ETHER  
The medium is Ethernet Bus.
- DL\_HDLC  
The medium is a bit synchronous communication line.
- DL\_CHAR  
The medium is a character synchronous communication line (for example, BISYNC).
- DL\_CTCA  
The medium is a channel-to-channel adapter.
- DL\_FDDI  
The medium is a Fiber Distributed Data Interface.
- DL\_FC  
The medium is Fiber Channel.
- DL\_ATM  
The medium is Asynchronous Transfer Mode.
- DL\_IPATM  
The medium is Internet IP over Asynchronous Transfer Mode, RFC 1577.
- DL\_X25  
The medium is X.25 LAPB.
- DL\_IPX25  
The medium is Internet IP over X.25, RFC 1356.
- DL\_ISDN  
The medium is ISDN.
- DL\_HIPPI  
The medium is High Performance Parallel Interface.
- DL\_100VG  
The medium is 100 Based VG Ethernet.
- DL\_100VGTPR  
The medium is 100 Based VG Token Ring.
- DL\_ETH\_CSMA  
The medium is capable of both ISO 8802/3 and Ethernet.
- DL\_100BT  
The medium is 100 Base T Ethernet.
- DL\_FRAME  
The medium is Frame Relay LAPF.

**DL\_MPFRAME**

The medium is Multi-Protocol over Frame Relay encapsulation, RFC 1490.

**DL\_ASYNC**

The medium is a bit asynchronous communication line.

**DL\_LOOP**

The medium is a software loopback.

**DL\_OTHER**

Any other medium that is not listed above.

*dl\_mac\_type* values above 0x80000000 are available for private or experimental use and will not be defined by this Specification.

*dl\_reserved*

is a reserved field whose value must be set to zero.

*dl\_current\_state*

conveys the state of the DLPI interface for the stream when the DLS provider issued this acknowledgement. See Appendix B for a list of DLPI states and an explanation of each.

*dl\_sap\_length*

indicates the current length of the SAP component of the DLSAP address. It may have a negative, zero or positive value. A positive value indicates the ordering of the SAP and PHYSICAL 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 service modes ( connection-mode,connectionless-mode or acknowledged connectionless-mode, or any combination of these modes) the DLS provider can support. It contains a bit-mask specifying one or more than one of the following values:

- DL\_CODLS    connection-oriented data link service
- DL\_CLDLS    connectionless data link service
- DL\_ACLDLS   acknowledged connectionless data link service.

Once a specific service mode has been bound to the stream, this field returns that specific service mode.

*dl\_qos\_length*

conveys the length, in bytes, of the negotiated/selected values of the quality of service (QOS) parameters. Section 5, Quality of Data Link Service , describes quality of service and its associated parameters completely. For connection-mode service, the returned values are those agreed during negotiation. For connectionless-mode service, the values are those currently selected by the DLS user. If quality of service 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 in connection-mode service, or those that will be applied to each data unit transmission in connectionless-mode service. If the DLS provider supports both connection-mode and connectionless-mode services but the DLS user has not yet bound a specific service mode, the DLS provider may return either connection-mode or connectionless-mode QOS parameter values.

The QOS values are conveyed in the structures defined in Section 4.4 on page 118. For any parameter the DLS provider does not support or cannot determine, the corresponding entry will be set to DL\_UNKNOWN. If the DLS provider does not support any QOS parameters, this length field will be set to zero.

*dl\_qos\_offset*

conveys the offset from the beginning of the M\_PCPROTO block where the current quality of service parameters begin.

*dl\_qos\_range\_length*

conveys the length, in bytes, of the available range of QOS parameter values supported by the DLS provider. For connection-mode service, this is the range available to the calling DLS user in a connect request. For connectionless-mode, this is the range available for each data unit transmission. If the DLS provider supports both connection-mode and connectionless-mode services but the DLS user has not yet bound a specific service mode, the DLS provider may return either connection-mode or connectionless-mode QOS parameter values.

The range of available QOS values is conveyed in the structures defined in Section 4.4 on page 118. For any parameter the DLS provider does not support or cannot determine, the corresponding entry will be set to DL\_UNKNOWN. If the DLS provider does not support any QOS parameters, this length field will be set to zero.

*dl\_qos\_range\_offset*

conveys the offset from the beginning of the M\_PCPROTO block where the available range of quality of service parameters begins.

*dl\_provider\_style*

conveys the style of DLS provider associated with the DLPI stream (see Section 1.4.1 on page 6). The following provider classes are defined:

- |           |   |
|-----------|---|
| DL_STYLE1 | The PPA is implicitly attached to the DLPI stream by opening the appropriate major/minor device number. |
| DL_STYLE2 | The DLS user must explicitly attach a PPA to the DLPI stream using DL_ATTACH_REQ.                       |

DLS users implemented in a protocol-independent manner must access this parameter to determine whether the DLS attach service must be invoked explicitly.

*dl\_addr\_offset*

conveys the 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*

indicates the current version of the dlpi that is supported. LI *dl\_brdcst\_addr\_length* indicates the length of the physical broadcast address.

*dl\_brdcst\_addr\_offset*

indicates the offset of the physical broadcast address from the beginning of the PCPROTO block.

*dl\_growth*

conveys a 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.

**NAME**

DL\_ATTACH\_REQ (*dl\_attach\_req\_t*) — requests the DLS provider associate a physical point of attachment (PPA) with astream. DL\_ATTACH\_REQ is needed for style 2 DLS providers to identify the physical medium over which communication will transpire. The request may not be issued to a style 1 DLS provider; doing so may cause errors.

The DLS provider may initialize the physical line on receipt of this primitive or the DL\_BIND\_REQ. Otherwise, the line must be initialized through some management mechanism before this request is issued by the DLS user. Either way, the physical link must be initialized and ready for use on successful completion of the DL\_BIND\_REQ.

**SYNOPSIS**

The message consists of one M\_PROTO message block, which contains the following structure.

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_ATTACH\_REQ.

*dl\_ppa*

conveys the identifier of the physical point of attachment to be associated with the stream. The format of the identifier is provider-specific, and it must contain sufficient information to distinguish the desired PPA from all possible PPAs on a system. At a minimum, this must include identification of the physical medium over which communication will transpire. For media that multiplex multiple channels over a single physical medium, this identifier should also specify a specific channel to be used for communication (where each channel on a physical medium is associated with a separate PPA).

Because of the provider-specific nature of this value, DLS user software that is to be protocol independent should avoid hard-coding the PPA identifier. The DLS user should retrieve the necessary PPA identifier from some other entity (such as a management entity) and insert it without inspection into the DL\_ATTACH\_REQ.

**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, message DL\_ERROR\_ACK is returned and the resulting state is unchanged.

**ERRORS**

DL\_ACCESS

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

DL\_BADPPA

The specified PPA is invalid.

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.

**NAME**

DL\_DETACH\_REQ (*dl\_detach\_req\_t*) — for style 2 DLS providers, this requests the DLS provider detach a physical point of attachment (PPA) from a stream. The request may not be issued to a style 1 DLS provider; doing so may cause errors.

**SYNOPSIS**

The message consists of one M\_PROTO message block, which contains the following structure.

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

**PARAMETERS**

*dl\_primitive*  
conveys 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, message DL\_ERROR\_ACK is returned and the resulting state is unchanged.

**ERRORS**

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.

**NAME**

DL\_BIND\_REQ (*dl\_bind\_req\_t*) — requests the DLS provider bind a DLSAP to the stream. The DLS user must identify the address of the DLSAP to be bound to the stream. For connection-mode service, 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.

A stream is viewed as active when the DLS provider may transmit and receive protocol data units destined to or originating from the stream. The PPA associated with each stream must be initialized upon completion of the processing of the DL\_BIND\_REQ (see Section 3.1.1 on page 31). More specifically, the DLS user is ensured that the PPA is initialized when the DL\_BIND\_ACK is received. If the PPA cannot be initialized, the DL\_BIND\_REQ will fail.

A stream may be bound as a "connection management" stream, such that it will receive all connect requests that arrive through a given PPA (see Section 1.5 on page 8). In this case, the *dl\_sap* will be ignored.

**SYNOPSIS**

The message consists of one M\_PROTO message block, which contains the following structure.

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

**PARAMETERS**

*dl\_primitive*  
conveys DL\_BIND\_REQ.

*dl\_sap*  
conveys sufficient information to identify the DLSAP that will be bound to the DLPI stream (see Section 1.4 on page 6 for a description of DLSAP addresses). The format of this information is specific to a given DLS provider, and may contain the full DLSAP address or some portion of that address sufficient to uniquely identify the DLSAP in question. The full address of the bound DLSAP will be returned in the DL\_BIND\_ACK.

The following rules are used by the DLS provider when binding a DLSAP address:

- The DLS provider must define and manage its DLSAP address space.
- DLPI allows the same DLSAP to be bound to multiple streams, but a given DLS provider may need to restrict its address space to allow one stream per DLSAP.
- The DLS provider may not be able to bind the specified DLSAP address for one of the following reasons:
  1. the DLS provider may statically associate a specific DLSAP with each stream
  2. the DLS provider may only support one stream per DLSAP and the DLS user attempted to bind a DLSAP that was already bound to another stream.

In case of reason 1, the value of *dl\_sap* is ignored by the DLS provider and the DL\_BIND\_ACK returns the DLSAP address that is already associated with the stream. In case of reason 2, if the DLS provider cannot bind the given DLSAP to the stream, it

may attempt to choose an alternate DLSAP and return that on the DL\_BIND\_ACK. If an alternate DLSAP cannot be chosen, the DLS provider will return a DL\_ERROR\_ACK and set *dl\_errno* to DL\_NOADDR.

Because of the provider-specific nature of the DLSAP address, DLS user software that is to be protocol independent should avoid hard-coding this value. The DLS user should retrieve the necessary DLSAP address from some other entity (such as a management entity or higher layer protocol entity) and insert it without inspection into the DL\_BIND\_REQ.

#### *dl\_max\_conind*

conveys the 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 (see Section 3.2.2 on page 59 for details on how this value is used to support multi-threaded connect processing). The DLS provider may not be able to support the value supplied in *dl\_max\_conind*, as specified by the following rules:

- If the provider cannot support the specified number of outstanding connect indications, it should set the value down to a number it can support.
- Only one stream that is bound to the indicated DLSAP may have an allowed number of maximum outstanding connect indications greater than zero. If a DL\_BIND\_REQ specifies a value greater than zero, but another stream has already bound itself to the DLSAP with a value greater than zero, the DLS provider will fail the request, setting *dl\_errno* to DL\_BOUND on the DL\_ERROR\_ACK.
- If a stream with *dl\_max\_conind* greater than zero is used to accept a connection, the stream will be found busy during the duration of the connection, and no other streams may be bound to the same DLSAP with a value of *dl\_max\_conind* greater than zero. This restriction prevents more than one stream bound to the same DLSAP from receiving connect indications and accepting connections. Accepting a connection on such a stream is only allowed if there is just a single outstanding connect indication being processed.
- A DLS user should always be able to request *dl\_max\_conind* value of zero, since this indicates to the DLS provider that the stream will only be used to originate connect requests.
- A stream with a negotiated value of *dl\_max\_conind* that is greater than zero may not originate connect requests.

This field is ignored in connectionless-mode service.

#### *dl\_service\_mode*

conveys the desired mode of service for this stream, and may contain one of the following:

- DL\_CODLS     connection-oriented data link service
- DL\_CLDLS     connectionless data link service
- DL\_ACLDLS    acknowledged connectionless data link service.

If the DLS provider does not support the requested service mode, a DL\_ERROR\_ACK will be generated, specifying DL\_UNSUPPORTED.

#### *dl\_conn\_mgmt*

if non-zero, indicates that the stream is the “connection management” stream for the PPA to which the stream is attached. When an incoming connect request arrives, the DLS provider will first look for a stream bound with *dl\_max\_conind* greater than zero that is

associated with the destination DLSAP. If such a stream is found, the connect indication will be issued on that stream. Otherwise, the DLS provider will issue the connect indication on the "connection management" stream for that PPA, if one exists. Only one "connection management" stream is allowed per PPA, so an attempt to bind a second connection management stream on a PPA will fail with the DLPI error set to DL\_BOUND. When *l\_conn\_mgmt* is non-zero, the value of *dl\_sap* will be ignored. In connectionless-mode service, *dl\_conn\_mgmt* is ignored by the DLS provider.

***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. The DLS Provider will not generate DL\_XID\_IND and/or DL\_TEST\_IND, and will error a DL\_XID\_REQ and/or DL\_TEST\_REQ. If the DLS Provider does not support automatic handling of XID and/or TEST responses, a DL\_ERROR\_ACK will be generated, specifying DL\_NOAUTO, DL\_NOXIDAUTO or DL\_NOTESTAUTO. If the Provider receives an XID or TEST request from the DLS User, a DL\_ERROR\_ACK will be generated specifying DL\_XIDAUTO or DL\_TESTAUTO respectively.

The *dl\_xidtest\_flg* contains a bit-mask specifying zero or more of the following values:

DL\_AUTO\_XID automatically respond to XID commands

DL\_AUTO\_TEST automatically respond to TEST commands.

**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, message DL\_ERROR\_ACK is returned and the resulting state is unchanged.

**ERRORS****DL\_ACCESS**

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

**DL\_BADADDR**

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

**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 "connectionmanagement" stream to a PPA.

**DL\_INITFAILED**

Automatic initialization of the PPA failed.

**DL\_NOTINIT**

The PPA had not been initialized prior to this request.

**DL\_NOADDR**

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

**DL\_NOAUTO**

Automatic handling of XID and TEST responses not supported.

DL\_NOTESTAUTO

Automatic handling of TEST response not supported.

DL\_NOXIDAUTO

Automatic handling of XID response not supported.

DL\_OUTSTATE

The primitive was issued from an invalid state.

DL\_SYSERR

DL\_UNSUPPORTED

The DLS provider does not support requested service mode on this stream. A system error has occurred and the UNIX system error is indicated in the DL\_ERROR\_ACK.

**NAME**

DL\_BIND\_ACK (*dl\_bind\_ack\_t*) — 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 to a DL\_BIND\_REQ.

**SYNOPSIS**

The message consists of one M\_PCPROTO message block, which contains the following structure.

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

**PARAMETERS***dl\_primitive*

conveys DL\_BIND\_ACK.

*dl\_sap*

conveys the DLSAP address information associated with the bound DLSAP. It corresponds to the *dl\_sap* field of the associated DL\_BIND\_REQ, which contains either part or all of the DLSAP address. For that portion of the DLSAP address conveyed in the DL\_BIND\_REQ, this field contains the corresponding portion of the address for the DLSAP that was actually bound.

*dl\_addr\_length*

conveys the length of the complete DLSAP address that was bound to the DLPI stream (see Section 1.4 on page 6 for a description of DLSAP addresses). The bound DLSAP is chosen according to the guidelines presented under the description of DL\_BIND\_REQ.

*dl\_addr\_offset*

conveys the offset from the beginning of the M\_PCPROTO block where the DLSAP address begins.

*dl\_max\_conind*

conveys the allowed, maximum number of outstanding DL\_CONNECT\_IND messages to be supported 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. The rules for negotiating this value are presented under the description of DL\_BIND\_REQ.

*dl\_xidtest\_flg*

conveys the XID and TEST responses supported by the provider.

DL\_AUTO\_XID XID response handled automatically

DL\_AUTO\_TEST TEST response handled automatically.

If no value is specified in *dl\_xidtest\_flg*, it indicates that automatic handling of XID and TEST responses is not supported by the Provider.

**STATE**

The message is valid in state DL\_BIND\_PENDING.

**NEW STATE**

The resulting state is DL\_IDLE.

**NAME**

*DL\_UNBIND\_REQ* (*dl\_unbind\_req\_t*) — requests the DLS provider to unbind the DLSAP that had been bound by a previous *DL\_BIND\_REQ* from this stream. If one or more DLSAPs were bound to the stream using a *DL\_SUBS\_BIND\_REQ*, and have not been unbound using a *DL\_SUBS\_UNBIND\_REQ*, the *DL\_UNBIND\_REQ* will unbind all the subsequent DLSAPs for that stream along with the DLSAP bound using the previous *DL\_BIND\_REQ*.

At the successful completion of the request, the DLS user may issue a new *DL\_BIND\_REQ* for a potentially new DLSAP.

**SYNOPSIS**

The message consists of one *M\_PROTO* message block, which contains the following structure.

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

**PARAMETERS**

*dl\_primitive*  
conveys *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, message *DL\_ERROR\_ACK* is returned and the resulting state is unchanged.

**ERRORS**

*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*.

**NAME**

DL\_SUBS\_BIND\_REQ (*dl\_subs\_bind\_req\_t*) — 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.

**SYNOPSIS**

The message consists of one M\_PROTO message block, which contains the following structure.

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_SUBS\_BIND\_REQ.

*dl\_subs\_sap\_offset*

conveys the offset of the DLSAP from the beginning of the M\_PROTO block.

*dl\_subs\_sap\_length*

conveys the 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. If the request fails, message DL\_ERROR\_ACK is returned and the resulting state is unchanged.

**ERRORS**

DL\_ACCESS

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

DL\_BADADDR

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

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\_TOOMANY**

Limit exceeded on the maximum number of DLSAPs per stream.

**DL\_UNSUPPORTED**

Requested addressing class not supported.

**NAME**

DL\_SUBS\_BIND\_ACK (*dl\_subs\_bind\_ack\_t*) — 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.

**SYNOPSIS**

The message consists of one M\_PCPROTO message block, which contains the following structure.

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_SUBS\_BIND\_ACK.

*dl\_subs\_sap\_offset*

conveys the offset of the DLSAP from the beginning of the M\_PCPROTO block.

*dl\_subs\_sap\_length*

conveys the length of the specified DLSAP.

**STATE**

The message is valid in state DL\_SUBS\_BIND\_PND

**NEW STATE**

The resulting state is DL\_IDLE.

**NAME**

DL\_SUBS\_UNBIND\_REQ (*dl\_subs\_unbind\_req\_t*) — requests the DLS Provider to unbind the DLSAP that had been bound by a previous DL\_SUBS\_BIND\_REQ from this stream.

**SYNOPSIS**

The message consists of one M\_PROTO message block, which contains the following structure:

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_SUBS\_UNBIND\_REQ.

*dl\_subs\_sap\_offset*

conveys the offset of the DLSAP from the beginning of the M\_PROTO block.

*dl\_subs\_sap\_length*

conveys the 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, message DL\_ERROR\_ACK is returned and the resulting state is unchanged.

**ERRORS**

DL\_BADADDR

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

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.

**NAME**

DL\_ENABMULTI\_REQ (*dl\_enabmulti\_req\_t*) — 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.

**SYNOPSIS**

The message consists of one M\_PROTO message block, which contains the following structure:

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_ENABMULTI\_REQ

*dl\_addr\_length*

conveys the length of the multicast address

*dl\_addr\_offset*

conveys the 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 acknowledgement 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, message DL\_ERROR\_ACK is returned and the resulting state is unchanged.

**ERRORS**

DL\_BADADDR

Address information was invalid or was in an incorrect format.

DL\_NOTSUPPORTED

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

DL\_OUTSTATE

The primitive was issued from an invalid state

DL\_TOOMANY

Too many multicast address enable attempts. Limit exceeded.

**NAME**

DL\_DISABMULTI\_REQ (*dl\_disabmulti\_req\_t*) — requests the DLS Provider to disable specific multicast addresses on a per Stream basis.

**SYNOPSIS**

The message consists of one M\_PROTO message block, which contains the following structure:

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_DISABMULTI\_REQ

*dl\_addr\_length*

conveys the length of the physical address

*dl\_addr\_offset*

conveys the 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 acknowledgement 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, message DL\_ERROR\_ACK is returned and the resulting state is unchanged.

**ERRORS**

DL\_BADADDR

Address information was invalid or in an incorrect format.

DL\_NOTENAB

Address specified is not enabled.

DL\_NOTSUPPORTED

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

DL\_OUTSTATE

The primitive was issued from an invalid state.

**NAME**

DL\_PROMISCON\_REQ (*dl\_promiscon\_req\_t*) — 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.

**SYNOPSIS**

The message consists of one M\_PROTO message block, which contains the following structure.

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_PROMISCON\_REQ

*dl\_level*

indicates promiscuous mode at the physical or SAP level:

DL\_PROMISC\_PHYS

indicates promiscuous mode at the physical level

DL\_PROMISC\_SAP

indicates promiscuous mode at the SAP level

DL\_PROMISC\_MULTI

indicates promiscuous mode for all multicast addresses.

**STATE**

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

**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.

**ERRORS**

DL\_NOTSUPPORTED

Primitive is known but not supported by the DLS Provider

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\_UNSUPPORTED

Requested service is not supplied by the provider.

**NAME**

DL\_PROMISCOFF\_REQ (*dl\_promiscoff\_req\_t*) — 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.

**SYNOPSIS**

The message consists of one M\_PROTO message block, which contains the following structure.

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_PROMISCOFF\_REQ

*dl\_level*

indicates promiscuous mode at the physical or SAP level:

DL\_PROMISC\_PHYS

indicates promiscuous mode at the physical level

DL\_PROMISC\_SAP

indicates promiscuous mode at the SAP level

DL\_PROMISC\_MULTI

indicates promiscuous mode for all multicast addresses.

**STATE**

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

**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.

**ERRORS**

DL\_NOTENAB

Mode not enabled.

DL\_NOTSUPPORTED

Primitive is known but not supported by the DLS Provider

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.

**NAME**

DL\_OK\_ACK (*dl\_ok\_ack\_t*) — 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 acknowledgement.

**SYNOPSIS**

The message consists of one M\_PCPROTO message block, which contains the following structure.

```
typedef struct {
    t_uscalar_t dl_primitive;
    t_uscalar_t dl_correct_primitive;
} dl_ok_ack_t;
```

**PARAMETERS**

*dl\_primitive*  
conveys DL\_OK\_ACK.

*dl\_correct\_primitive*  
identifies the successfully received primitive that is being acknowledged.

**STATE**

The message is valid in response to any of the following:

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\_DISABMULTI\_REQ or  
DL\_PROMISCOFF\_REQ

from any of several states as defined in Appendix B.

**NEW STATE**

The resulting state depends on the current state and is defined fully in Appendix B.

**NAME**

DL\_ERROR\_ACK (*dl\_error\_ack\_t*) — informs the DLS user that a previously issued request or response was invalid. It conveys the identity of the primitive in error, a DLPI error code, and if appropriate, a UNIX system error code. Whenever this primitive is generated, it indicates that the DLPI state is identical to what it was before the erroneous request or response.

**SYNOPSIS**

The message consists of one M\_PCPROTO message block, which contains the following structure.

```
typedef struct {
    t_uscalar_t dl_primitive;
    t_uscalar_t dl_error_primitive;
    t_uscalar_t dl_errno;
    t_uscalar_t dl_unix_errno;
} dl_error_ack_t;
```

**PARAMETERS**

*dl\_primitive*  
conveys DL\_ERROR\_ACK.

*dl\_error\_prim*  
identifies the primitive in error.

*dl\_errno*  
conveys the DLPI error code associated with the failure. See the individual request or response for the error codes that are applicable. In addition to those errors:

DL\_BADPRIM  
error is returned if an unrecognized primitive is issued by the DLS user.

DL\_NOTSUPPORTED  
error is returned if an unsupported primitive is issued by the DLS user.

*dl\_unix\_errno*  
conveys the 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.

## 3.2 Connection-mode Service Primitives

### 3.2.1 Scope

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

### 3.2.2 Multi-threaded Connection Establishment

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 (single-threaded), 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. This capability is desirable, for example, when imposing a priority scheme on all DLS users attempting to establish a connection. The DLS provider will pass all connect indications to the called DLS user (up to some preestablished 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. The DLS user can also receive a `DL_DISCONNECT_IND` with a correlation value when the calling DLS user or the DLS provider abort a 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`. Thus, the lifetime of a correlation value is the duration of the connection establishment phase, and as good programming practice it should not be used for any other purpose by the DLS provider.

The DLS provider assigns the correlation value for each connect indication. Correlation values must be unique among all outstanding connect indications on a given stream. The values may, but need not, be unique across all streams to the DLS provider. The correlation value must be a positive, non-zero value. There is no implied sequencing of connect indications using the correlation value; the values do not have to increase sequentially for each new connect indication.

**3.2.3 List of Connection-mode Service Primitives**

The connection-mode service primitives are listed below and are defined in the following man-pages:

- DL\_CONNECT\_REQ
- DL\_CONNECT\_IND
- DL\_CONNECT\_RES
- DL\_CONNECT\_CON
- DL\_TOKEN\_REQ
- DL\_TOKEN\_ACK
- DL\_DATA\_REQ
- DL\_DATA\_IND
- DL\_DISCONNECT\_REQ
- DL\_DISCONNECT\_IND
- DL\_RESET\_REQ
- DL\_RESET\_IND
- DL\_RESET\_RES
- DL\_RESET\_CON

**NAME**

DL\_CONNECT\_REQ (*dl\_connect\_req\_t*) — requests the DLS provider establish a data link connection with a remote DLS user. The request contains the DLSAP address of the remote (called) DLS user and quality of service parameters to be negotiated during connection establishment.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below.

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

**PARAMETERS**

LI *dl\_primitive*  
conveys DL\_CONNECT\_REQ.

*dl\_dest\_addr\_length*  
conveys the length of the DLSAP address that identifies the DLS user with whom a connection is to be established. If the called user is implemented using DLPI, this address is the full DLSAP address returned on the DL\_BIND\_ACK.

*dl\_dest\_addr\_offset*  
conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_qos\_length*  
conveys the length of the quality of service (QOS) parameter values desired by the DLS user initiating a connection. The desired QOS values are conveyed in the appropriate structure defined in Section 4.4 on page 118. A full specification of these QOS parameters and rules for negotiating their values is presented in Chapter 4.

If the DLS user does not wish to specify a particular QOS value, the value DL\_QOS\_DONT\_CARE may be specified. If the DLS user does not care to specify any QOS parameter values, this field may be set to zero.

*dl\_qos\_offset*  
conveys the 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 message 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 connect request is rejected by the called DLS user, the called DLS user cannot be reached, or the DLS provider and/or called DLS user do not agree on the specified quality of service, a DL\_DISCONNECT\_IND is sent to the calling DLS user, resulting in state DL\_IDLE.

If the request is erroneous, message DL\_ERROR\_ACK is returned and the resulting state is unchanged.

**ERRORS****DL\_ACCESS**

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

**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\_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.

**NAME**

DL\_CONNECT\_IND (*dl\_connect\_ind\_t*) — conveys to the local DLS user that a remote (calling) DLS user wishes to establish a data link connection. The indication contains the DLSAP address of the calling and called DLS user, and the quality of service parameters as specified by the calling DLS user and negotiated by the DLS provider.

The DL\_CONNECT\_IND also contains a number that allows the DLS user to correlate a subsequent DL\_CONNECT\_RES, DL\_DISCONNECT\_REQ, or DL\_DISCONNECT\_IND with the indication (see Section 3.2.2 on page 59).

The number of outstanding DL\_CONNECT\_IND primitives issued by the DLS provider must not exceed the value of *dl\_max\_conind* as returned on the DL\_BIND\_ACK. If this limit is reached and an additional connect request arrives, the DLS provider must not pass the corresponding connect indication to the DLS user until a response is received for an already outstanding indication.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below.

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

**PARAMETERS**

*dl\_primitive*  
conveys DL\_CONNECT\_IND.

*dl\_correlation*  
conveys the 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. This value, then, enables the DLS user to multi-thread connect indications and responses. All outstanding connect indications must have a distinct, non-zero correlation value set by the DLS provider.

*dl\_called\_addr\_length*  
conveys the length of the address of the DLSAP for which this DL\_CONNECT\_IND primitive is intended. This address is the full DLSAP address specified by the calling DLS user and is typically the value returned on the DL\_BIND\_ACK associated with the given stream.

*dl\_called\_addr\_offset*  
conveys the offset from the beginning of the M\_PROTO message block where the called DLSAP address begins.

*dl\_calling\_addr\_length*  
conveys the length of the address of the DLSAP from which the DL\_CONNECT\_REQ primitive was sent.

*dl\_calling\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block where the calling DLSAP address begins.

*dl\_qos\_length*

conveys the range of quality of service parameter values desired by the calling DLS user and negotiated by the DLS provider. The range of QOS values is conveyed in the appropriate structure defined in Section 4.4 on page 118. A full specification of these QOS parameters and rules for negotiating their values is presented in Chapter 4. For any parameter the DLS provider does not support or cannot determine, the corresponding parameter values will be set to DL\_UNKNOWN. If the DLS provider does not support any QOS parameters, this length field will be set to zero.

*dl\_qos\_offset*

conveys the 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 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.

**NAME**

DL\_CONNECT\_RES (*dl\_connect\_res\_t*) — 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. The response contains the correlation number from the corresponding DL\_CONNECT\_IND, selected quality of service parameters, and an indication of the stream on which to accept the connection.

After issuing this primitive, the DLS user may immediately begin transferring data using the DL\_DATA\_REQ primitive. If the DLS provider receives one or more DL\_DATA\_REQ primitives from the local DLS user before it has completed connection establishment, however, it must queue the data transfer requests internally until the connection is successfully established.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below.

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

**PARAMETERS***dl\_primitive*

conveys DL\_CONNECT\_RES.

*dl\_correlation*

conveys the 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, conveys the token associated with the responding stream on which the DLS provider is to establish the connection; this stream must be in the state DL\_IDLE. The token value for a stream can be obtained by issuing a DL\_TOKEN\_REQ on that stream. If the DLS user is accepting the connection on the stream where the connect indication arrived, this value must be zero. See Section 1.3.1.2 on page 4 for a description of the connection response model.

*dl\_qos\_length*

conveys the length of the quality of service parameter values selected by the called DLS user.

The selected QOS values are conveyed in the appropriate structure as defined in Section 4.4 on page 118. A full specification of these QOS parameters and rules for negotiating their values is presented in Chapter 4. If the DLS user does not care which value is selected for a particular QOS parameter, the value DL\_QOS\_DONT\_CARE may be specified. If the DLS user does not care which values are selected for all QOS parameters, this field may be set to zero.

*dl\_qos\_offset*

conveys the 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\_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\_resp\_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.

**ERRORS****DL\_ACCESS**

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

**DL\_BADCORR**

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

**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\_OUTSTATE**

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

**DL\_PENDING**

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

**DL\_SYSERR**

A system error has occurred and the UNIX system error is indicated in the DL\_ERROR\_ACK.

**NAME**

DL\_CONNECT\_CON (*dl\_connect\_con\_t*) — informs the local DLS user that the requested data link connection has been established. The primitive contains the DLSAP address of the responding DLS user and the quality of service parameters as selected by the responding DLS user.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below.

```
typedef struct {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_resp_addr_length;
    t_uscalar_t  dl_resp_addr_offset;
    t_uscalar_t  dl_qos_length;
    t_uscalar_t  dl_qos_offset;
    t_uscalar_t  dl_growth;
} dl_connect_con_t;
```

**PARAMETERS***dl\_primitive*

conveys DL\_CONNECT\_CON.

*dl\_resp\_addr\_length*

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

*dl\_resp\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block where the responding DLSAP address begins.

*dl\_qos\_length*

conveys the length of the quality of service parameter values selected by the responding DLS user. The selected QOS values are conveyed in the appropriate structure defined in Section 4.4 on page 118. A full specification of these QOS parameters and rules for negotiating their values is presented in Chapter 4. For any parameter the DLS provider does not support or cannot determine, the corresponding parameter value will be set to DL\_UNKNOWN. If the DLS provider does not support any QOS parameters, this length field will be set to zero.

*dl\_qos\_offset*

conveys the 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 will be set to zero.

**STATE**

The message is valid in state DL\_OUTCON\_PENDING.

**NEW STATE**

The resulting state is DL\_DATAXFER.

## NAME

DL\_TOKEN\_REQ (*dl\_token\_req\_t*) — 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.

## SYNOPSIS

The message consists of one M\_PCPROTO message block, which contains the following structure.

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

## PARAMETERS

*dl\_primitive*  
conveys DL\_TOKEN\_REQ.

## STATE

The message is valid in any state in which a local acknowledgement is not pending, as described in Appendix B.

## RESPONSE

The DLS provider responds to the information request with a DL\_TOKEN\_ACK.

**NAME**

DL\_TOKEN\_ACK (*dl\_token\_ack\_t*) — this message is sent in response to DL\_TOKEN\_REQ; it conveys the connection response token assigned to the stream.

**SYNOPSIS**

The message consists of one M\_PCPROTO message block, which contains the following structure.

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

**PARAMETERS**

*dl\_primitive*  
conveys DL\_TOKEN\_ACK.

*dl\_token*  
conveys the 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.

**NAME**

DL\_DATA\_REQ — conveys a complete DLSDU from the DLS user to the DLS provider for transmission over the data link connection.

The DLS provider guarantees to deliver each DLSDU to the remote DLS user in the same order as received from the local DLS user. If the DLS provider detects unrecoverable data loss during data transfer, this may be indicated to the DLS user by a DL\_RESET\_IND, or by a DL\_DISCONNECT\_IND (if the connection is lost).

**SYNOPSIS**

The message consists of one or more M\_DATA message blocks containing at least one byte of data.

To simplify support of a *read/write* interface to the data link layer, the DLS provider must recognize and process messages that consist of one or more M\_DATA message blocks with no preceding M\_PROTO message block. This message type may originate from the *write()* system call<sup>3</sup>.

**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.

**NOTE****Support of Direct User-Level Access**

A STREAMS module would implement “more” field processing itself to support direct user-level access. This module could collect messages and send them in one larger message to the DLS provider, or break large DLSDUs passed to the DLS user into smaller messages. The module would only be pushed if the DLS user was a user-level process.

---

3. This does not imply that DLPI will directly support a pure *read/write*. If such an interface is desired, a STREAMS module could be implemented to be pushed above the DLS provider.

**NAME**

DL\_DATA\_IND — conveys a DLSDU from the DLS provider to the DLS user. The DLS provider guarantees to deliver each DLSDU to the local DLS user in the same order as received from the remote DLS user. If the DLS provider detects unrecoverable data loss during data transfer, this may be indicated to the DLS user by a DL\_RESET\_IND, or by a DL\_DISCONNECT\_IND (if the connection is lost).

**SYNOPSIS**

The message consists of one or more M\_DATA blocks containing at least one byte of data.

**STATE**

The message is valid in state DL\_DATA\_XFER.

**NEW STATE**

The resulting state is unchanged.

**NAME**

DL\_DISCONNECT\_REQ (*dl\_disconnect\_req\_t*) — 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 disconnect.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below.

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

**PARAMETERS**

*dl\_primitive*  
conveys DL\_DISCONNECT\_REQ.

*dl\_reason*  
conveys the reason for the disconnect:

**Reason for Disconnect:**

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\_DISC\_UNSPECIFIED  
reason unspecified.

*dl\_correlation*  
if non-zero, conveys the correlation number that was contained in the DL\_CONNECT\_IND being rejected (see Section 3.2.2 on page 59). This value permits the DLS provider to associate the primitive with the proper DL\_CONNECT\_IND when rejecting an incoming connection. If the 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.

**RESPONSE**

If the disconnect is successful, DL\_OK\_ACK is sent to the DLS user resulting in state DL\_IDLE.

If the request fails, message DL\_ERROR\_ACK is returned, and the resulting state is unchanged.

**ERRORS**

**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.

**NAME**

DL\_DISCONNECT\_IND (*dl\_disconnect\_ind\_t*) — 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. The primitive indicates the origin and the cause of the disconnect.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below.

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

**PARAMETERS***dl\_primitive*

conveys DL\_DISCONNECT\_IND.

*dl\_originator*

conveys whether the disconnect was DLS user or DLS provider originated (DL\_USER or DL\_PROVIDER, respectively).

*dl\_reason*

conveys the reason for the disconnect:

**Reason for Disconnect**

DL\_DISC\_PERMANENT\_CONDITION  
connection released due to permanent condition

DL\_DISC\_TRANSIENT\_CONDITION  
connection released due to transient condition

DL\_CONREJ\_DEST\_UNKNOWN  
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\_DISC\_UNSPECIFIED  
reason unspecified

*dl\_correlation*

if non-zero, conveys the correlation number that was contained in the DL\_CONNECT\_IND that is being aborted (see Section 3.2.2 on page 59). 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* will 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.

**NAME**

**DL\_RESET\_REQ** (*dl\_reset\_req\_t*) — requests that the DLS provider initiate the resynchronization 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.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below.

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

**PARAMETERS**

*dl\_primitive*  
conveys DL\_RESET\_REQ.

**STATE**

The message is valid in state DL\_DATAXFER.

**NEW STATE**

The resulting state is DL\_USER\_RESET\_PENDING.

**RESPONSE**

There is no immediate response to the reset request. However, as resynchronization completes, DL\_RESET\_CON is sent to the initiating DLS user, resulting in state DL\_DATAXFER. If the request fails, message DL\_ERROR\_ACK is returned and the resulting state is unchanged.

**ERRORS****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.

**NAME**

DL\_RESET\_IND (*dl\_reset\_ind\_t*) — informs the DLS user that either the remote DLS user is resynchronizing 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.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below.

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_RESET\_IND.

*dl\_originator*

conveys whether the reset was originated by the DLS user or DLS provider (DL\_USER or DL\_PROVIDER, respectively) .

*dl\_reason*

conveys the reason for the reset.

**Reason for 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 resynchronization 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.

**NAME**

DL\_RESET\_RES (*dl\_reset\_res\_t*) — directs the DLS provider to complete resynchronizing the data link connection.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below.

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

**PARAMETERS**

*dl\_primitive*  
conveys 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.

**ERRORS**

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.

**NAME**

DL\_RESET\_CON (*dl\_reset\_con\_t*) — informs the reset-initiating DLS user that the reset has completed.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below.

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

**PARAMETERS**

*dl\_primitive*  
conveys DL\_RESET\_CON.

**STATE**

The message is valid in state DL\_USER\_RESET\_PENDING.

**NEW STATE**

The resulting state is DL\_DATAXFER.

### **3.3 Connectionless-mode Service Primitives**

#### **3.3.1 Scope**

This section describes the primitives that support the connectionless-mode service of the data link layer. These primitives support the connectionless data transfer service described earlier.

#### **3.3.2 List of Connectionless-mode Service Primitives**

The connectionless-mode service primitives are listed below and are defined in the following man-pages:

- DL\_UNITDATA\_REQ
- DL\_UNITDATA\_IND
- DL\_UDERROR\_IND
- DL\_UDQOS\_REQ

**NAME**

DL\_UNITDATA\_REQ (*dl\_unitdata\_req\_t*) — 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 DLSDU. It is the responsibility of the DLS user to do any necessary sequencing or retransmission of DLSDU in the event of a presumed loss.

**SYNOPSIS**

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 {
    t_uscalar_t          dl_primitive;
    t_uscalar_t          dl_dest_addr_length;
    t_uscalar_t          dl_dest_addr_offset;
    dl_priority_t        dl_priority;
} dl_unitdata_req_t;
```

**PARAMETERS**

*dl\_primitive*  
conveys DL\_UNITDATA\_REQ.

*dl\_dest\_addr\_length*  
conveys the 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*  
conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_priority*  
indicates the 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, message 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.

**ERRORS**

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**

The primitive was issued from an invalid state.

**DL\_UNSUPPORTED**

Requested priority not supplied by provider.

**NAME**

DL\_UNITDATA\_IND (*dl\_unitdata\_ind\_t*) — conveys one DLSDU from the DLS provider to the DLS user.

**SYNOPSIS**

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 {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
    t_uscalar_t  dl_src_addr_length;
    t_uscalar_t  dl_src_addr_offset;
    t_uscalar_t  dl_group_address;
} dl_unitdata_ind_t;
```

**PARAMETERS***dl\_primitive*

conveys DL\_UNITDATA\_IND.

*dl\_dest\_addr\_length*

conveys the length of the address of the DLSAP where this DL\_UNITDATA\_IND is intended to be delivered.

*dl\_dest\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_src\_addr\_length*

conveys the length of the DLSAP address of the sending DLS user.

*dl\_src\_addr\_offset*

conveys the 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 state DL\_IDLE.

**NEW STATE**

The resulting state is unchanged.

**NAME**

DL\_UDERROR\_IND (*dl\_uderror\_ind\_t*) — 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.

**SYNOPSIS**

The message consists of either one M\_PROTO message block or one M\_PCPROTO message block containing the structure shown below.

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

**PARAMETERS***dl\_primitive*

conveys DL\_UDERROR\_IND.

*dl\_dest\_addr\_length*

conveys the length of the DLSAP address of the destination DLS user.

*dl\_dest\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_unix\_errno*

conveys the 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*

conveys the DLPI error code associated with the failure. See **ERRORS** 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 (for example, 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.

**NAME**

DL\_UDQOS\_REQ (*dl\_udqos\_req\_t*) — requests the DLS provider to apply the specified quality of service parameter values to subsequent data unit transmissions. These new values will remain in effect until another DL\_UDQOS\_REQ is issued.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below.

```
typedef struct {
    t_uscalar_t dl_primitive;
    t_uscalar_t dl_qos_length;
    t_uscalar_t dl_qos_offset;
} dl_udqos_req_t;
```

**PARAMETERS**

*dl\_primitive*  
conveys DL\_UDQOS\_REQ.

*dl\_qos\_length*  
conveys the length, in bytes, of the requested quality of service parameter values. The values are conveyed in the appropriate structure defined in Section 4.4 on page 118. The available range of QOS values that may be selected is specified by the *dl\_qos\_range\_length* and *dl\_qos\_range\_offset* parameters in the DL\_INFO\_ACK primitive. For any parameter whose value the DLS user does not wish to select, the value DL\_QOS\_DONT\_CARE may be set and the DLS provider will maintain the current value for that parameter. See Chapter 4. for a full description of the quality of service parameters.

*dl\_qos\_offset*  
conveys the offset from the beginning of the M\_PROTO message block where the quality of service parameters begin.

**STATE**

The message is valid in state DL\_IDLE.

**NEW STATE**

The resulting state is DL\_UDQOS\_PENDING.

**RESPONSE**

If the quality of service request is successful, DL\_OK\_ACK is sent to the DLS user and the resulting state is DL\_IDLE. If the request fails, message DL\_ERROR\_ACK is returned and the resulting state is unchanged.

**ERRORS**

DL\_BADQOSPARAM  
The quality of service parameters contained values outside the range of those supported by the DLS provider.

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

DL\_OUTSTATE  
The primitive was issued from an invalid state.

### 3.4 Primitives to handle XID and TEST operations

#### 3.4.1 Scope

This section describes the service primitives that support the XID and TEST operations. The DLS User can issue these primitives to the DLS Provider requesting the provider to send an XID or a TEST frame. On receipt of an XID or TEST frame from the remote side, the DLS Provider can send the appropriate indication to the User.

#### 3.4.2 List of Primitives Handling XID and TEST operations

The primitives which handle XID and TEST operations are listed below and are defined in the following man-pages:

- DL\_TEST\_REQ
- DL\_TEST\_IND
- DL\_TEST\_RES
- DL\_TEST\_CON
- DL\_XID\_REQ
- DL\_XID\_IND
- DL\_XID\_RES
- DL\_XID\_CON

**NAME**

DL\_TEST\_REQ (*dl\_test\_req\_t*) — conveys one TEST command DLSDU from the DLS User to the DLS Provider for transmission to a peer DLS Provider.

**SYNOPSIS**

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 {
    t_uscalar_t dl_primitive;
    t_uscalar_t dl_flag;
    t_uscalar_t dl_dest_addr_length;
    t_uscalar_t dl_dest_addr_offset;
} dl_test_req_t;
```

**PARAMETERS**

*dl\_primitive*

conveys DL\_TEST\_REQ

*dl\_flag*

indicates flag values for the request as follows:

DL\_POLL\_FINAL

indicates if the poll/final bit is set.

*dl\_dest\_addr\_length*

conveys the 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*

conveys the 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.

**ERRORS**

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\_OUTSTATE

The primitive was issued from an invalid state

DL\_TESTAUTO

Previous bind request specified automatic handling of TEST responses.

DL\_UNSUPPORTED

Requested service not supplied by provider.

**NAME**

DL\_TEST\_IND (*dl\_test\_ind\_t*) — conveys the TEST response/indication DLSDU from the DLS Provider to the DLS User.

**SYNOPSIS**

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 {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_flag;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
    t_uscalar_t  dl_src_addr_length;
    t_uscalar_t  dl_src_addr_offset;
} dl_test_ind_t;
```

**PARAMETERS**

*dl\_primitive*

conveys DL\_TEST\_IND

*dl\_flag*

indicates the flag values associated with the received TEST frame:

DL\_POLL\_FINAL

indicates if the poll/final bit is set.

*dl\_dest\_addr\_length*

conveys the 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*

conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_src\_addr\_length*

conveys the 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*

conveys the 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.

**NAME**

DL\_TEST\_RES (*dl\_test\_res\_t*) — conveys the TEST response DLSDU from the DLS User to the DLS Provider in response to a DL\_TEST\_IND.

**SYNOPSIS**

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 {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_flag;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
} dl_test_res_t;
```

**PARAMETERS**

*dl\_primitive*

conveys DL\_TEST\_RES

*dl\_flag*

indicates the flag values for the response as follows:

DL\_POLL\_FINAL

indicates if the poll/final bit is set.

*dl\_dest\_addr\_length*

conveys the 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*

conveys the 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.

**NAME**

DL\_TEST\_CON (*dl\_test\_con\_t*) — conveys the TEST response DLSDU from the DLS Provider to the DLS User in response to a DL\_TEST\_REQ.

**SYNOPSIS**

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 {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_flag;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
    t_uscalar_t  dl_src_addr_length;
    t_uscalar_t  dl_src_addr_offset;
} dl_test_con_t;
```

**PARAMETERS**

*dl\_primitive*

conveys DL\_TEST\_RES

*dl\_flag*

indicates the flag values for the request as follows:

DL\_POLL\_FINAL

indicates if the poll/final bit is set.

*dl\_dest\_addr\_length*

conveys the 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*

conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_src\_addr\_length*

conveys the 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*

conveys the 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.

**NAME**

DL\_XID\_REQ (*dl\_xid\_req\_t*) — conveys one XID DLSDU from the DLS User to the DLS Provider for transmission to a peer DLS User.

**SYNOPSIS**

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 {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_flag;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
} dl_xid_req_t;
```

**PARAMETERS**

*dl\_primitive*

conveys DL\_XID\_REQ

*dl\_flag*

indicates the 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*

conveys the 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*

conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

**STATE**

The message is valid in state 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.

**ERRORS**

DL\_BADADDR

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

DL\_BADDATA

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

DL\_NOTSUPPORTED

Primitive is known but not supported by the DLS Provider

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\_XIDAUTO**

Previous bind request specified Provider would handle XID.

**NAME**

DL\_XID\_IND (*dl\_xid\_ind\_t*) — conveys an XID DLSDU from the DLS Provider to the DLS User.

**SYNOPSIS**

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 {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_flag;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
    t_uscalar_t  dl_src_addr_length;
    t_uscalar_t  dl_src_addr_offset;
} dl_xid_ind_t;
```

**PARAMETERS**

*dl\_primitive*

conveys DL\_XID\_IND

*dl\_flag*

conveys the 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*

conveys the 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*

conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_src\_addr\_length*

conveys the length of the source DLSAP address. If the source user is implemented using DLPI, this address if the full DLSAP address returned on the DL\_BIND\_ACK.

*dl\_src\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block where the source DLSAP address begins.

**STATE**

The message is valid in state DL\_IDLE and DL\_DATAXFER.

**NEW STATE**

The resulting state is unchanged.

**RESPONSE**

The DLS User must respond with a DL\_XID\_RES.

**NAME**

DL\_XID\_RES (*dl\_xid\_res\_t*) — conveys an XID DLSDU from the DLS User to the DLS Provider in response to a DL\_XID\_IND.

**SYNOPSIS**

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 {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_flag;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
} dl_xid_res_t;
```

**PARAMETERS**

*dl\_primitive*

conveys DL\_XID\_RES

*dl\_flag*

conveys the flag values associated with the received XID frame.

DL\_POLL\_FINAL

indicates if the poll/final bit is set.

*dl\_dest\_addr\_length*

conveys the 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*

conveys the 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.

**NAME**

DL\_XID\_CON (*dl\_xid\_con\_t*) — conveys an XID DLSDU from the DLS Provider to the DLS User in response to a DL\_XID\_REQ.

**SYNOPSIS**

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 {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_flag;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
    t_uscalar_t  dl_src_addr_length;
    t_uscalar_t  dl_src_addr_offset;
} dl_xid_con_t;
```

**PARAMETERS**

*dl\_primitive*

conveys DL\_XID\_CON

*dl\_flag*

conveys the flag values associated with the received XID frame.

DL\_POLL\_FINAL

indicates if the poll/final bit is set.

*dl\_dest\_addr\_length*

conveys the 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*

conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_src\_addr\_length*

conveys the 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*

conveys the 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.

## 3.5 Acknowledged Connectionless-mode Service Primitives

### 3.5.1 Scope

This section describes the primitives that support the acknowledged connectionless-mode service of the data link layer. These primitives support the acknowledged connectionless data transfer service described earlier.

### 3.5.2 List of Acknowledged Connectionless-mode Service Primitives

The acknowledged connectionless-mode service primitives are listed below and are defined in the following man-pages:

- DL\_DATA\_ACK\_REQ
- DL\_DATA\_ACK\_IND
- DL\_DATA\_ACK\_STATUS\_IND
- DL\_REPLY\_REQ
- DL\_REPLY\_IND
- DL\_REPLY\_STATUS\_IND
- DL\_REPLY\_UPDATE\_REQ
- DL\_REPLY\_UPDATE\_STATUS\_IND

**NAME**

DL\_DATA\_ACK\_REQ (*dl\_data\_ack\_req\_t*) — this request is passed to the Data Link Provider to request that a DLSDU be sent to a peer DLS User using acknowledged connectionless mode data unit transmission procedures.

**SYNOPSIS**

Consists of one M\_PROTO message block containing the structure shown below, followed by one or more M\_DATA blocks containing one or more bytes 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 {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_correlation;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
    t_uscalar_t  dl_src_addr_length;
    t_uscalar_t  dl_src_addr_offset;
    t_uscalar_t  dl_priority;
    t_uscalar_t  dl_service_class;
} dl_data_ack_req_t;
```

**PARAMETERS***dl\_primitive*

conveys DL\_DATA\_ACK\_REQ

*dl\_correlation*

conveys a unique identifier which will be returned in the DL\_DATA\_ACK\_STATUS\_IND primitive to allow the DLS User to correlate the status to the appropriate DL\_DATA\_ACK\_REQ primitive.

*dl\_dest\_addr\_length*

conveys the 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*

conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_src\_addr\_length*

conveys the length of the DLSAP address of the source DLS User.

*dl\_src\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block where the source DLSAP address begins.

*dl\_priority*

indicates the priority value within the supported range for this particular DLSDU.

*dl\_service\_class*

specifies whether or not an acknowledge capability in the medium access control sublayer is to be used for the data unit transmission.

## DL\_RQST\_RSP

request acknowledgement service from the medium access control sublayer if supported

**DL\_RQST\_NORSP**

no acknowledgement service requested from the medium access control sublayer.

**STATE**

This message is valid in state DL\_IDLE.

**NEW STATE**

The resulting state is unchanged.

**RESPONSE**

If the request is erroneous, message DL\_ERROR\_ACK is returned, and the resulting state is unchanged. If the DLS Provider accepts the data for transmission, a DL\_DATA\_ACK\_STATUS\_IND is returned. This indication will indicate the success or failure of the data transmission. Although the exchange service is connectionless, in-sequence delivery is guaranteed for data sent by the initiating station.

**ERRORS****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\_NOTSUPPORTED**

Primitive is valid, but not supported.

**DL\_OUTSTATE**

The primitive was issued from an invalid state.

**DL\_UNSUPPORTED**

Requested service or priority not supported by Provider (Request with response at the Medium Access Control sublayer).

**NAME**

DL\_DATA\_ACK\_IND (*dl\_data\_ack\_ind\_t*) — conveys one DLSDU from the DLS Provider to the DLS User. This primitive indicates the arrival of a non-null, non-duplicate DLSDU from a peer Data Link User entity.

**SYNOPSIS**

Consists of one M\_PROTO message block containing the structure shown below, followed by one or more M\_DATA blocks containing one or more bytes 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 {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
    t_uscalar_t  dl_src_addr_length;
    t_uscalar_t  dl_src_addr_offset;
    t_uscalar_t  dl_priority;
    t_uscalar_t  dl_service_class;
} dl_data_ack_ind_t;
```

**PARAMETERS***dl\_primitive*

conveys DL\_DATA\_ACK\_IND

*dl\_dest\_addr\_length*

conveys the length of the DLSAP address of the destination DLS User. If the destination user is implemented using DLPI, this address is the full DLSAP

*dl\_dest\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_src\_addr\_length*

conveys the length of the DLSAP address of the source DLS User.

*dl\_src\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block where the source DLSAP address begins. address returned on the DL\_BIND\_ACK.

*dl\_priority*

priority provided for the data unit transmission.

*dl\_service\_class*

specifies whether or not an acknowledge capability in the medium access control sublayer is to be used for the data unit transmission.

## DL\_RQST\_RSP

use acknowledgement service in the medium access control sublayer.

## DL\_RQST\_NORSP

no acknowledgement service to be used in the medium access control sublayer.

**STATE**

This message is valid in state DL\_IDLE.

**NEW STATE**

The resulting state is unchanged.

**NAME**

DL\_DATA\_ACK\_STATUS\_IND (*dl\_data\_ack\_status\_ind\_t*) — conveys the results of the previous associated DL\_DATA\_ACK\_REQ from the DLS Provider to the DLS User.

**SYNOPSIS**

Consists of one M\_PROTO message block containing the structure shown below.

```
typedef struct {
    t_uscalar_t dl_primitive;
    t_uscalar_t dl_correlation;
    t_uscalar_t dl_status;
} dl_data_ack_status_ind_t;
```

**PARAMETERS***dl\_primitive*

conveys DL\_DATA\_ACK\_STATUS\_IND

*dl\_correlation*

conveys the unique identifier passed with the DL\_DATA\_ACK\_REQ primitive, to allow the DLS User correlate the status to the appropriate DL\_DATA\_ACK\_REQ.

*dl\_status*

indicates the success or failure of the previous associated acknowledged connectionless-mode data unit transmission request.

## DL\_CMD\_OK

command accepted.

## DL\_CMD\_RS

unimplemented or inactivated service.

## DL\_CMD\_UE

LLC User Interface error

## DL\_CMD\_PE

protocol error

## DL\_CMD\_IP

permanent implementation dependent error

## DL\_CMD\_UN

resources temporarily unavailable.

## DL\_CMD\_IT

temporary implementation dependent error.

**STATE**

This message is valid in state DL\_IDLE.

**NEW STATE**

The resulting state is unchanged.

**NAME**

DL\_REPLY\_REQ (*dl\_reply\_req\_t*) — this request primitive is passed to the DLS Provider by the DLS User to request that a DLSDU be returned from a peer DLS Provider or that DLSDUs be exchanged between stations using acknowledged connectionless mode data unit exchange procedures.

**SYNOPSIS**

Consists of one M\_PROTO message block containing the structure shown below, followed by one or more M\_DATA blocks with one or more bytes of data.

```
typedef struct {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_correlation;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
    t_uscalar_t  dl_src_addr_length;
    t_uscalar_t  dl_src_addr_offset;
    t_uscalar_t  dl_priority;
    t_uscalar_t  dl_service_class;
} dl_reply_req_t;
```

**PARAMETERS***dl\_primitive*

conveys DL\_REPLY\_REQ

*dl\_correlation*

conveys a unique identifier which will be returned in the DL\_REPLY\_STATUS\_IND primitive to allow the DLS User to correlate the status to the appropriate DL\_REPLY\_REQ primitive.

*dl\_dest\_addr\_length*

conveys the 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*

conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_src\_addr\_length*

conveys the length of the DLSAP address of the source DLS User.

*dl\_src\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block where the source DLSAP address begins.

*dl\_priority*

priority provided for the data unit transmission.

*dl\_service\_class*

specifies whether or not an acknowledge capability in the medium access control sublayer is to be used for the data unit transmission.

**STATE**

This primitive is valid in state DL\_IDLE.

**NEW STATE**

The resulting state is unchanged.

**RESPONSE**

If the request is erroneous, message DL\_ERROR\_ACK is returned, and the resulting state is unchanged. If the message is valid, a DL\_REPLY\_STATUS\_IND is returned. This will indicate the success or failure of the previous associated acknowledged connectionless-mode data unit exchange.

**ERRORS****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\_NOTSUPPORTED**

Primitive is valid, but not supported.

**DL\_OUTSTATE**

The primitive was issued from an invalid state.

**DL\_UNSUPPORTED**

Requested service not supported by Provider (Request with response at the Medium Access Control sublayer).

**NAME**

DL\_REPLY\_IND (*dl\_reply\_ind\_t*) — this primitive is the service indication primitive for the acknowledged connectionless-mode data unit exchange service. It is passed from the DLS Provider to the DLS User to indicate either a successful request of a DLSDU from the peer data link user entity, or exchange of DLSDU with a peer data link user entity.

**SYNOPSIS**

Consists of one M\_PROTO message block containing the structure shown below, followed by zero or more M\_DATA blocks.

```
typedef struct {
    t_uscalar_t  dl_primitive;
    t_uscalar_t  dl_dest_addr_length;
    t_uscalar_t  dl_dest_addr_offset;
    t_uscalar_t  dl_src_addr_length;
    t_uscalar_t  dl_src_addr_offset;
    t_uscalar_t  dl_priority;
    t_uscalar_t  dl_service_class;
} dl_reply_ind_t;
```

**PARAMETERS**

*dl\_primitive*

conveys DL\_REPLY\_IND

*dl\_dest\_addr\_length*

conveys the 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*

conveys the offset from the beginning of the M\_PROTO message block where the destination DLSAP address begins.

*dl\_src\_addr\_length*

conveys the length of the DLSAP address of the source DLS User.

*dl\_src\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block where the source DLSAP address begins.

*dl\_priority*

priority provided for the data unit transmission.

*dl\_service\_class*

specifies whether or not an acknowledge capability in the medium access control sublayer is to be used for the data unit transmission.

**STATE**

This primitive is valid in state DL\_IDLE.

**NEW STATE**

The resulting state is unchanged.

**NAME**

DL\_REPLY\_STATUS\_IND (*dl\_reply\_status\_ind\_t*) — this indication primitive is passed from the DLS Provider to the DLS User to indicate the success or failure of the previous associated acknowledged connectionless mode data unit exchange request.

**SYNOPSIS**

Consists of one M\_PROTO message block containing the structure shown below, followed by zero or more M\_DATA blocks.

```
typedef struct {
    t_uscalar_t dl_primitive;
    t_uscalar_t dl_correlation;
    t_uscalar_t dl_status;
} dl_reply_status_ind_t;
```

**PARAMETERS***dl\_primitive*

conveys DL\_REPLY\_STATUS\_IND

*dl\_correlation*

conveys the unique identifier passed with the DL\_REPLY\_REQ primitive, to allow the DLS User correlate the status to the appropriate DL\_REPLY\_REQ.

*dl\_status*

Indicates the success or failure of the previous associated acknowledged connectionless-mode data unit exchange request.

## DL\_CMD\_OK

command accepted.

## DL\_CMD\_RS

unimplemented or inactivated service.

## DL\_CMD\_UE

LLC User Interface error

## DL\_CMD\_PE

protocol error

## DL\_CMD\_IP

permanentimplementation dependent error

## DL\_CMD\_UN

resources temporarily available.

## DL\_CMD\_IT

temporary implementation dependent error.

## DL\_RSP\_OK

response DLSDU present.

## DL\_RSP\_RS

unimplemented or inactivated service.

## DL\_RSP\_NE

response DLSDU never submitted.

## DL\_RSP\_NR

response DLSDU not requested.

DL\_RSP\_UE  
LLC User interface error.

DL\_RSP\_IP  
permanent implementation dependent error.

DL\_RSP\_UN  
resources temporarily unavailable.

DL\_RSP\_IT  
temporary implementation dependent error.

**STATE**

This primitive is valid in state DL\_IDLE.

**NEW STATE**

The resulting state is unchanged.

**NAME**

DL\_REPLY\_UPDATE\_REQ (*dl\_reply\_update\_req\_t*) — conveys a DLSDU to the DLS Provider from the DLS User to be held by the DLS Provider and sent out at a later time when requested to do so by the peer DLS Provider.

**SYNOPSIS**

Consists of one M\_PROTO message block containing the structure shown below, followed by one or more M\_DATA blocks.

```
typedef struct {
    t_uscalar_t dl_primitive;
    t_uscalar_t dl_correlation;
    t_uscalar_t dl_src_addr_length;
    t_uscalar_t dl_src_addr_offset;
} dl_reply_update_req_t;
```

**PARAMETERS**

*dl\_primitive*

conveys DL\_REPLY\_UPDATE\_REQ

*dl\_correlation*

conveys context specific information to be returned in the DL\_REPLY\_UPDATE\_STATUS\_IND primitive to allow the DLS User correlate the status to the appropriate previous request.

*dl\_src\_addr\_length*

conveys the length of the DLSAP address of the source DLS User.

*dl\_src\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block where the source DLSAP address begins.

**STATE**

This primitive is valid in state DL\_IDLE.

**NEW STATE**

The resulting state is unchanged.

**RESPONSE**

If the request is erroneous, a DL\_ERROR\_ACK is returned with the appropriate error code. Otherwise, a DL\_REPLY\_UPDATE\_STATUS\_IND is returned, which indicates the success or failure of the DL\_REPLY\_UPDATE\_REQ.

**ERRORS**

DL\_BADDATA

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

DL\_NOTSUPPORTED

Primitive is known, but not supported.

DL\_OUTSTATE

The primitive was issued from an invalid state.

**NAME**

DL\_REPLY\_UPDATE\_STATUS\_IND (*dl\_reply\_update\_status\_ind\_t*) — this primitive is the service confirmation primitive for the reply data unit preparation service. This primitive is sent to the DL User from the DLS Provider to indicate the success or failure of the previous associated data unit preparation request.

**SYNOPSIS**

Consists of one M\_PROTO message block containing the structure shown below.

```
typedef struct {
    t_uscalar_t dl_primitive;
    t_uscalar_t dl_correlation;
    t_uscalar_t dl_status;
} dl_reply_update_req_t;
```

**PARAMETERS***dl\_primitive*

conveys DL\_UPDATE\_STATUS\_IND

*dl\_correlation*

Indicates the context information passed with the DL\_REPLY\_UPDATE\_REQ to allow the DLS User correlate the status with the appropriate previous request.

*dl\_status*

indicates the success or failure of the previous associated data unit preparation request.

## DL\_CMD\_OK

command accepted.

## DL\_CMD\_RS

unimplemented or inactivated service.

## DL\_CMD\_UE

LLC User Interface error

## DL\_CMD\_PE

protocol error

## DL\_CMD\_IP

permanent implementation dependent error

## DL\_CMD\_UN

resources temporarily available.

## DL\_CMD\_IT

temporary implementation dependent error.

## DL\_RSP\_OK

response DLSDU present.

## DL\_RSP\_RS

unimplemented or inactivated service.

## DL\_RSP\_NE

response DLSDU never submitted.

## DL\_RSP\_NR

response DLSDU not requested.

DL\_RSP\_UE  
LLC User interface error.

DL\_RSP\_IP  
permanent implementation dependent error.

DL\_RSP\_UN  
resources temporarily unavailable.

DL\_RSP\_IT  
temporary implementation dependent error.

**STATE**

This primitive is valid in state DL\_IDLE.

**NEW STATE**

The resulting state is unchanged.

## Quality of Data Link Service

### 4.1 Characteristics

The quality of data link service is defined by the term "Quality of Service" (QOS), and describes certain characteristics of transmission between two DLS users. These characteristics are attributable solely to the DLS provider, but are observable by the DLS users. The visibility of QOS characteristics enables a DLS user to determine, and possibly negotiate, the characteristics of transmission needed to communicate with the remote DLS user.

### 4.2 Overview of Quality of Service

Quality of service characteristics apply to both the connection and connectionless modes of service. The semantics for each mode are discussed below.

#### 4.2.1 Connection-mode Service

"Quality of Service" (QOS) refers to certain characteristics of a data link connection as observed between the connection endpoints. QOS describes the specific aspects of a data link connection that are attributable to the DLS provider.

QOS is defined in terms of QOS parameters. The parameters give DLS users a means of specifying their needs. These parameters are divided into two groups, based on how their values are determined:

- QOS parameters that are negotiated on a per-connection basis during connection establishment
- QOS parameters that are not negotiated during connection establishment. The values are determined or known through other methods, usually administrative.

The QOS parameters that can be negotiated during connection establishment are:

- throughput
- transit delay
- priority
- protection.

The QOS parameters for throughput and transit delay are negotiated end-to-end between the two DLS users and the DLS provider. The QOS parameters for priority and protection are negotiated locally by each DLS user with the DLS provider. The QOS parameters that cannot be negotiated are residual error rate and resilience. Section 4.5 on page 121 describes the rules for QOS negotiation.

Once the connection is established, the agreed QOS values are not renegotiated at any point. There is no guarantee by any DLS provider that the original QOS values will be maintained, and the DLS users are not informed if QOS changes. The DLS provider also need only record those QOS values selected at connection establishment for return in response to the DL\_INFO\_REQ primitive. Quality of Data Link Service for Connectionless-mode and Acknowledged Connectionless-mode Service.

#### **4.2.2 QOS for Connectionless/Acknowledged Connectionless**

The QOS for connectionless-mode and acknowledged connectionless-mode service refers to characteristics of the data link layer between two DLSAPs, attributable to the DLS provider. The QOS applied to each DL\_UNITDATA\_REQ/DL\_DATA\_ACK\_REQ primitive may be independent of the QOS applied to preceding and following DL\_UNITDATA\_REQ/DL\_DATA\_ACK\_REQ primitives. QOS cannot be negotiated between two DLS users as in the connection-mode service.

Every DL\_UNITDATA\_REQ/DL\_DATA\_ACK\_REQ primitive may have certain QOS values associated with it. The supported range of QOS parameter values is made known to the DLS user in response to the DL\_INFO\_REQ primitive. The DLS user may select specific QOS parameter values to be associated with subsequent data unit transmissions using the DL\_UDQOS\_REQ primitive. This selection is a strictly local management function. If different QOS values are to be associated with each transmission, DL\_UDQOS\_REQ may be issued to alter those values before each DL\_UNITDATA\_REQ or DL\_DATA\_ACK\_REQ is issued.

### 4.3 QOS Parameter Definitions

This section describes the quality of service parameters supported by DLPI for both connection-mode and connectionless-mode services. The following table summarizes the supported parameters. It indicates to which service mode (connection, connectionless, or both) the parameter applies. For those parameters supported by the connection-mode service, the table also indicates whether the parameter value is negotiated during connection establishment. If so, the table further indicates whether the QOS values are negotiated end-to-end among both DLS users and the DLS provider, or locally for each DLS user independently with the DLS provider.

Parameter	Service Mode	Negotiation
throughput	connection	end-to-end
transit delay	both	end-to-end
priority	both	local
protection	both	local
residual error rate	both	none
resilience	connection	none

**Table 4-1** QOS Supported Parameters

#### 4.3.1 Throughput

Throughput is a connection-mode QOS parameter that has end-to-end significance. It is defined as the total number of DLSDU bits successfully transferred by a DL\_DATA\_REQ/DL\_DATA\_IND primitive sequence divided by the input/output time, in seconds, for that sequence. Successful transfer of a DLSDU is defined to occur when the DLSDU is delivered to the intended user without error, in proper sequence, and before connection termination by the receiving DLS user.

The input/output time for a DL\_DATA\_REQ/DL\_DATA\_IND primitive sequence is the greater of both:

- the time between the first and last DL\_DATA\_REQ in a sequence
- the time between the first and last DL\_DATA\_IND in the sequence.

Throughput is only meaningful for a sequence of complete DLSDUs.

Throughput is specified and negotiated for the transmit and receive directions independently at connection establishment. The throughput specification defines the target and minimum acceptable values for a connection. Each specification is an average rate.

The DLS user can delay the receipt or sending of DLSDUs. The delay caused by a DLS user is not included in calculating the average throughput values.

##### Parameter Format

```
typedef struct {
    t_scalar_t    dl_target_value;
    t_scalar_t    dl_accept_value;
} dl_through_t;
```

This typedef is used to negotiate the transmit and receive throughput values.

**Parameter Definitions***dl\_target\_value*

specifies the desired throughput value for the connection in bits/second.

*dl\_accept\_value*

specifies the minimum acceptable throughput value for the connection in bits/second.

**4.3.2 Transit Delay**

Connection and connectionless modes can specify a transit delay, which indicates the elapsed time between a DL\_DATA\_REQ or DL\_UNITDATA\_REQ primitive and the corresponding DL\_DATA\_IND or DL\_UNITDATA\_IND primitive. The elapsed time is only computed for DLSDUs successfully transferred, as described previously for throughput.

In connection mode, transit delay is negotiated on an end-to-end basis during connection establishment. For each connection, transit delay is negotiated for the transmit and receive directions separately by specifying the target value and maximum acceptable value. For connectionless-mode service, a DLS user selects a particular value within the supported range using the DL\_UDQOS\_REQ primitive, and the value may be changed for each DLSDU submitted for connectionless transmission.

The transit delay for an individual DLSDU may be increased if the receiving DLS user flow controls the interface. The average and maximum transit delay values exclude any DLS user flow control of the interface. The values are specified in milliseconds, and assume a DLSDU size of 128 octets.

**Parameter Format**

```
typedef struct {
    t_scalar_t    dl_target_value;
    t_scalar_t    dl_accept_value;
} dl_transdelay_t;
```

This typedef is used to negotiate the transmit and receive transit delay values.

**Parameter Definitions***dl\_target\_value*

specifies the desired transit delay value.

*dl\_accept\_value*

specifies the maximum acceptable transit delay value.

**4.3.3 Priority**

Priority is negotiated locally between each DLS user and the DLS provider in connection-mode service, and can also be specified for connectionless-mode service. The specification of priority is concerned with the relationship between connections or the relationship between connectionless data transfer requests.

The parameter specifies the relative importance of a connection with respect to:

- the order in which connections are to have their QOS degraded, if necessary
- the order in which connections are to be released to recover resources, if necessary.

For connectionless-mode service, the parameter specifies the relative importance of unit data objects with respect to gaining use of shared resources.

For connection-mode service, each DLS user negotiates a particular priority value with the DLS provider during connection establishment. The value is specified by a minimum and a maximum within a given range. For connectionless-mode service, a DLS user selects a particular priority value within the supported range using the DL\_UDQOS\_REQ primitive, and the value may be changed for each DLSDU submitted for connectionless transmission.

This parameter only has meaning in the context of some management entity or structure able to judge relative importance. The priority has local significance only, with a value of zero being the highest priority and 100 being the lowest priority.

#### Parameter Format

```
typedef struct {
    t_scalar_t    dl_min;
    t_scalar_t    dl_max;
} dl_priority_t;
```

#### Parameter Definitions

*dl\_min*

specifies the minimum acceptable priority.

*dl\_max*

specifies the maximum desired priority.

#### 4.3.4 Protection

Protection is negotiated locally between each DLS user and the DLS provider in connection-mode service, and can also be specified for connectionless-mode service. Protection is the extent to which a DLS provider attempts to prevent unauthorized monitoring or manipulation of DLS user-originated information.

Protection is specified by a minimum and maximum protection option within the following range of possible protection options:

DL\_NONE

DLS provider will not protect any DLS user data

DL\_MONITOR

DLS provider will protect against passive monitoring

DL\_MAXIMUM

DLS provider will protect against modification, replay, addition, or deletion of DLS user data.

For connection-mode service, each DLS user negotiates a particular value with the DLS provider during connection establishment. The value is specified by a minimum and a maximum within a given range. For connectionless-mode service, a DLS user selects a particular value within the supported range using the DL\_UDQOS\_REQ primitive, and the value may be changed for each DLSDU submitted for connectionless transmission. Protection has local significance only.

**Parameter Format**

```
typedef struct {
    t_scalar_t    dl_min;
    t_scalar_t    dl_max;
} dl_protect_t;
```

**Parameter Definitions***dl\_min*

specifies the minimum acceptable protection.

*dl\_max*

specifies the maximum desired protection.

**4.3.5 Residual Error Rate**

Residual error rate (RER) is the ratio of total incorrect, lost and duplicated DLSDUs to the total DLSDUs transferred between DLS users during a period of time. The relationship between these quantities is defined below:

$$RER = \frac{DLSDU_l + DLSDU_i + DLSDU_e}{DLSDU_{tot}}$$

where

$DLSDU_{tot}$  = total DLSDUs transferred, which is the total of  $DLSDU_l$ ,  $DLSDU_i$ ,  $DLSDU_e$ , and correctly received DLSDUs

$DLSDU_e$  = DLSDUs received 2 or more times

$DLSDU_i$  = incorrectly received DLSDUs

$DLSDU_l$  = DLSDUs sent, but not received.

**Parameter Format**

```
t_scalar_t    l_residual_error;
```

The residual error value is scaled by a factor of 1,000,000, since the parameter is stored as a `t_scalar_t` integer in the QOS data structures. Residual error rate is not a negotiated QOS parameter. Its value is determined by procedures outside the definition of DLPI. It is assumed to be set by an administrative mechanism, which is informed of the value by network management.

**4.3.6 Resilience**

Resilience is meaningful in connection mode only, and represents the probability of either: DLS provider-initiated disconnects or DLS provider-initiated resets during a time interval of 10,000 seconds on a connection.

Resilience is not a negotiated QOS parameter. Its value is determined by procedures outside the definition of DLPI. It is assumed to be set by an administrative mechanism, which is informed of the value by network management.

**Parameter Format**

```
typedef struct {  
    t_scalar_t    dl_disc_prob;  
    t_scalar_t    dl_reset_prob;  
} dl_resilience_t;
```

**Parameter Definitions***dl\_disc\_prob*

specifies the probability of receiving a provider-initiated disconnect, scaled by 10000.

*dl\_reset\_prob*

specifies the probability of receiving a provider-initiated reset, scaled by 10000.

## 4.4 QOS Data Structures

To simplify the definition of the primitives containing QOS parameters and the discussion of QOS negotiation, the QOS parameters are organized into four structures. This section defines the structures and indicates which structures apply to which primitives.

Each structure is tagged with a type field contained in the first four bytes of the structure, similar to the tagging of primitives. The type field has been defined because of the current volatility of QOS parameter definition within the international standards bodies. If new QOS parameter sets are defined in the future for the data link layer, the type field will enable DLPI to accommodate these sets without breaking existing DLS user or provider implementations. However, DLS user and provider software should be cognizant of the possibility that new QOS structure types may be defined in future issues of the DLPI specification. If a DLS provider receives a structure type that it does not understand in a given primitive, the error `DL_BADQOSTYPE` should be returned to the DLS user in a `DL_ERROR_ACK` primitive.

Currently the following QOS structure types are defined:

`DL_QOS_CO_RANGE1`

QOS range structure for connection-mode service for Issue 1 of DLPI

`DL_QOS_CO_SEL1`

QOS selection structure for connection-mode service for Issue 1 of DLPI

`DL_QOS_CL_RANGE1`

QOS range structure for connectionless-mode service for Issue 1 of DLPI

`DL_QOS_CL_SEL1`

QOS selection structure for connectionless-mode service for Issue 1 of DLPI.

The syntax and semantics of each structure type is presented in the remainder of this section.

### 4.4.1 Structure `DL_QOS_CO_RANGE1`

Structure type `DL_QOS_CO_RANGE1` enables a DLS user and DLS provider to pass between them a range of QOS parameter values in the connection-mode service. The format of this structure type is:

```
typedef struct {
    t_uscalar_t          dl_qos_type;
    dl_throughput_t      dl_rcv_throughput;
    dl_transdelay_t      dl_rcv_trans_delay;
    dl_throughput_t      dl_xmt_throughput;
    dl_transdelay_t      dl_xmt_trans_delay;
    dl_priority_t        dl_priority;
    dl_protect_t         dl_protection;
    t_scalar_t           dl_residual_error;
    dl_resilience_t     dl_resilience;
} dl_qos_co_range1_t;
```

where the value of `dl_qos_type` is `DL_QOS_CO_RANGE1`. The fields of this structure correspond to the parameters defined in Section 4.3 on page 113. The throughput and transit delay parameters are specified for each direction of transmission on a data link connection.

This structure type is returned in the `dl_qos_range_length` and `dl_qos_range_offset` fields of the `DL_INFO_ACK`, and specifies the supported ranges of service quality supported by the DLS provider. In other words, it specifies the available range of QOS parameter values that may be specified on a `DL_CONNECT_REQ`.

For the DL\_CONNECT\_REQ and DL\_CONNECT\_IND primitives, this structure specifies the negotiable range of connection-mode QOS parameter values. See Section 4.5 on page 121 for the semantics of this structure in these primitives.

#### 4.4.2 Structure DL\_QOS\_CO\_SEL1

Structure type DL\_QOS\_CO\_SEL1 conveys selected QOS parameter values for connection-mode service between the DLS user and DLS provider. The format of this structure type is:

```
typedef struct {
    t_uscalar_t      dl_qos_type;
    t_scalar_t       dl_rcv_throughput;
    t_scalar_t       dl_rcv_trans_delay;
    t_scalar_t       dl_xmt_throughput;
    t_scalar_t       dl_xmt_trans_delay;
    t_scalar_t       dl_priority;
    t_scalar_t       dl_protection;
    t_scalar_t       dl_residual_error;
    dl_resilience_t dl_resilience;
} dl_qos_co_sel1_t;
```

where the value of *dl\_qos\_type* is DL\_QOS\_CO\_SEL1. The fields of this structure correspond to the parameters defined in Section 4.3 on page 113. The throughput and transit delay parameters are specified for each direction of transmission on a data link connection.

This structure type is returned in the *dl\_qos\_length* and *dl\_qos\_offset* fields of the DL\_INFO\_ACK, and specifies the current or default QOS parameter values associated with a stream. Default values are returned prior to connection establishment, and currently negotiated values are returned when a connection is active on the stream.

The structure type is used in the DL\_CONNECT\_RES to enable the responding DLS user to select particular QOS parameter values from the available range. The DL\_CONNECT\_CON primitive returns the selected values to the calling DLS user in this structure. See for the semantics of this structure in these primitives.

#### 4.4.3 Structure DL\_QOS\_CL\_RANGE1

Structure type DL\_QOS\_CL\_RANGE1 enables a DLS user and DLS provider to pass between them a range of QOS parameter values in the connectionless-mode service. The format of this structure type is:

```
typedef struct {
    t_uscalar_t      dl_qos_type;
    dl_transdelay_t  dl_trans_delay;
    dl_priority_t    dl_priority;
    dl_protect_t     dl_protection;
    t_scalar_t       dl_residual_error;
} dl_qos_cl_range1_t;
```

where the value of *dl\_qos\_type* is DL\_QOS\_CL\_RANGE1. The fields of this structure correspond to the parameters defined in

This structure type is returned in the *dl\_qos\_range\_length* and *dl\_qos\_range\_offset* fields of the DL\_INFO\_ACK, and specifies the range of connectionless-mode QOS parameter values supported by the DLS provider on the stream. The DLS user may select specific values from this range using the DL\_UDQOS\_REQ primitive, as described in Section 4.5 on page 121.

#### 4.4.4 Structure DL\_QOS\_CL\_SEL1

Structure type DL\_QOS\_CL\_SEL1 conveys selected QOS parameter values for connectionless-mode service between the DLS user and DLS provider. The format of this structure type is:

```
typedef struct {
    t_uscalar_t  dl_qos_type;
    t_scalar_t   dl_trans_delay;
    t_scalar_t   dl_priority;
    t_scalar_t   dl_protection;
    t_scalar_t   dl_residual_error;
} dl_qos_cl_sel1_t;
```

where the value of *dl\_qos\_type* is DL\_QOS\_CL\_SEL1. The fields of this structure correspond to the parameters defined in Section 4.3 on page 113. This structure type is returned in the *dl\_qos\_length* and *dl\_qos\_\_offset* fields of the DL\_INFO\_ACK, and specifies the current or default QOS parameter values associated with a stream. Default values are returned until the DLS user issues a DL\_UDQOS\_REQ to change the values, after which the currently selected values will be returned. The structure type is also used in the DL\_UDQOS\_REQ primitive to enable a DLS user to select particular QOS parameter values from the supported range, as described in Section 4.5 on page 121.

## 4.5 Procedures for QOS Negotiation and Selection

This section describes the methods used for negotiating and/or selecting QOS parameter values. In the connection-mode service, some QOS parameter values may be negotiated during connection establishment. For connectionless-mode service, parameter values may be selected for subsequent data transmission.

Throughout this section, two special QOS values are referenced. These are defined for all the parameters used in QOS negotiation and selection. The values are:

### DL\_UNKNOWN

This value indicates that the DLS provider does not know the value for the field or does not support that parameter.

### DL\_QOS\_DONT\_CARE

This value indicates that the DLS user does not care to what value the QOS parameter is set.

These values are used to distinguish between DLS providers that support and negotiate QOS parameters and those that cannot. The following sections include the interpretation of these values during QOS negotiation and selection.

### 4.5.1 Connection-mode QOS Negotiation

The current connection-mode QOS parameters can be divided into three types as follows:

- those that are negotiated end-to-end between peer DLS users and the DLS provider during connection establishment (throughput and transit delay)
- those that are negotiated locally between each DLS user and the DLS provider during connection establishment (priority and protection)
- those that cannot be negotiated (residual error rate and resilience).

The rules for processing these three types of parameters during connection establishment are described in this section.

The current definition of most existing data link protocols does not describe a mechanism for negotiating QOS parameters during connection establishment. As such, DLPI does not require every DLS provider implementation to support QOS negotiation. If a given DLS provider implementation cannot support QOS negotiation, two alternatives are available:

- The DLS provider may specify that any or all QOS parameters are unknown. This is indicated to the DLS user in the DL\_INFO\_ACK, where the values in the QOS range field (indicated by *dl\_qos\_range\_length* and *dl\_qos\_range\_offset*) and the current QOS field (indicated by *dl\_qos\_length* and *dl\_qos\_offset*) of this primitive are set to DL\_UNKNOWN. This value will also be indicated on the DL\_CONNECT\_IND and DL\_CONNECT\_CON primitives. If the DLS provider does not support any QOS parameters, the QOS length field may be set to zero in each of these of these primitives.
- The DLS provider may interpret QOS parameters with strictly local significance, and their values in the DL\_CONNECT\_IND primitive will be set to DL\_UNKNOWN.

A DLS user need not select a specific value for each QOS parameter. The special QOS parameter value, DL\_QOS\_DONT\_CARE, is used if the DLS user does not care what quality of service is provided for a particular parameter. The negotiation procedures presented below explain the exact semantics of this value during connection establishment.

If QOS parameters are supported by the DLS provider, the provider will define a set of default QOS parameter values that are used whenever DL\_QOS\_DONT\_CARE is specified for a QOS parameter value. These default values can be defined for all DLS users or can be defined on a

per DLS user basis. The default parameter value set is returned in the QOS field (indicated by *dl\_qos\_length* and *dl\_qos\_offset* ) of the DL\_INFO\_ACK before a DLS user negotiates QOS parameter values.

DLS provider addendum documentation must describe the known ranges of support for the QOS parameters and the default values, and also specify whether they are used in a local manner only.

The following procedures are used to negotiate QOS parameter values during connection establishment:

1. The DL\_CONNECT\_REQ specifies the DLS user's desired range of QOS values in the *dl\_qos\_co\_range1\_t* structure. The target and least-acceptable values are specified for throughput and transit delay, as described in Section 4.3.1 on page 113, and Section 4.3.2 on page 114. The target value is the value desired by the calling DLS user for the QOS parameters. The least acceptable value is the lowest value the calling user will accept. These values are specified separately for both the transmit and receive directions of the connection.

If either value is set to DL\_QOS\_DONT\_CARE the DLS provider will supply a default value, subject to the following consistency constraints:

- If DL\_QOS\_DONT\_CARE is specified for the target value, the value chosen by the DLS provider may not be less than the least-acceptable value.
- If DL\_QOS\_DONT\_CARE is specified for the least-acceptable value, the value set by the DLS provider cannot be greater than the target value.
- If DL\_QOS\_DONT\_CARE is specified for both the target and least-acceptable value, the DLS provider is free to select any value, without constraint, for the target and least acceptable values.

For priority and protection, the DL\_CONNECT\_REQ specifies a minimum and maximum desired value as defined in Section 4.3.3 on page 114 and Section 4.3.4 on page 115. As with throughput and transit delay, the DLS user may specify a value of DL\_QOS\_DONT\_CARE for either the minimum or maximum value. The DLS provider will interpret this value subject to the following consistency constraints:

- If DL\_QOS\_DONT\_CARE is specified for the maximum value, the value chosen by the DLS provider may not be less than the minimum value.
- If DL\_QOS\_DONT\_CARE is specified for the minimum value, the value set by the DLS provider cannot be greater than the maximum value.
- If DL\_QOS\_DONT\_CARE is specified for both the minimum and maximum values, the DLS provider is free to select any value, without constraint, for the maximum and minimum values.

The values of the residual error rate and resilience parameters in the DL\_CONNECT\_REQ have no meaning and are ignored by the DLS provider.

If the value of *dl\_qos\_length* in the DL\_CONNECT\_REQ is set to zero by the DLS user, the DLS provider should treat all QOS parameter values as if they were set to DL\_QOS\_DONT\_CARE, selecting any value in its supported range.

If the DLS provider cannot support throughput, transit delay, priority, and protection values within the ranges specified in the DL\_CONNECT\_REQ, a DL\_DISCONNECT\_IND should be sent to the calling DLS user.

2. If the requested ranges of values for throughput and transit delay in the DL\_CONNECT\_REQ are acceptable to the DLS provider, the QOS parameters will be adjusted to values the DLS provider will support. Only the target value may be adjusted, and it is set to a value the DLS provider is willing to provide (which may be of lower QOS than the target value). The least-acceptable value cannot be modified. The updated QOS range is then sent to the called DLS user in the *dl\_qos\_co\_range1\_t* structure of the DL\_CONNECT\_IND, where it is interpreted as the available range of service.

If the requested range of values for priority and protection in the DL\_CONNECT\_REQ is acceptable to the DLS provider, an appropriate value within the range is selected and saved for each parameter; these selected values will be returned to the DLS user in the corresponding DL\_CONNECT\_CON primitive. Because priority and protection are negotiated locally, the DL\_CONNECT\_IND will not contain values selected during negotiation with the calling DLS user. Instead, the DLS provider will offer a range of values in the DL\_CONNECT\_IND that will be supported locally for the called DLS user.

The DLS provider will also include the supported values for residual error rate and resilience in the DL\_CONNECT\_IND that is passed to the called DLS user.

If the DLS provider does not support negotiation of throughput, transit delay, priority, or protection, a value of DL\_UNKNOWN should be set in the least-acceptable, target, minimum, and maximum value fields of the DL\_CONNECT\_IND. Also, if the DLS provider does not support any particular QOS parameter, DL\_UNKNOWN should be specified in all value fields for that parameter. If the DLS provider does not support any QOS parameters, the value of *dl\_qos\_length* may be set to zero in the DL\_CONNECT\_IND.

3. Upon receiving the DL\_CONNECT\_IND, the called DLS user examines the QOS parameter values and selects a specific value from the proffered range of the throughput, transit delay, priority, and protection parameters. If the called DLS user does not agree on values in the given range, the connection should be refused with a DL\_DISCONNECT\_REQ primitive. Otherwise, the selected values are returned to the DLS provider in the *dl\_qos\_co\_sel1\_t* structure of the DL\_CONNECT\_RES primitive.

The values of residual error rate and resilience in the DL\_CONNECT\_RES are ignored by the DLS provider. These parameters may not be negotiated by the called DLS user. The selected values of throughput and transit delay are meaningful, however, and are adopted for the connection by the DLS provider. Similarly, the selected priority and protection values are adopted with local significance for the called DLS user.

If the user specifies DL\_QOS\_DONT\_CARE for either throughput, transit delay, priority, or protection on the DL\_CONNECT\_RES, the DLS provider will select a value from the range specified for that parameter in the DL\_CONNECT\_IND primitive. Also, a value of zero in the *dl\_qos\_length* field of the DL\_CONNECT\_RES is equivalent to DL\_QOS\_DONT\_CARE for all QOS parameters.

4. Upon completion of connection establishment, the values of throughput and transit delay as selected by the called DLS user are returned to the calling DLS user in the *dl\_qos\_co\_sel1\_t* structure of the DL\_CONNECT\_CON primitive. The values of priority and protection that were selected by the DLS provider from the range indicated in the DL\_CONNECT\_REQ will also be returned in the DL\_CONNECT\_CON. This primitive will also contain the values of residual error rate and resilience associated with the newly established connection. The DLS provider also saves the negotiated QOS parameter values for the connection, so that they may be returned in response to a DL\_INFO\_REQ primitive.

As with DL\_CONNECT\_IND, if the DLS provider does not support negotiation of throughput, transit delay, priority, or protection, a value of DL\_UNKNOWN should be

returned in the selected value fields. Furthermore, if the DLS provider does not support any particular QOS parameter, DL\_UNKNOWN should be specified in all value fields for that parameter, or the value of *dl\_qos\_length* may be set to zero in the DL\_CONNECT\_CON primitive.

#### 4.5.2 Connectionless-mode QOS Selection

This section describes the procedures for selecting QOS parameter values that will be associated with the transmission of connectionless data or acknowledged connectionless data.

As with connection-mode protocols, the current definition of most existing (acknowledged) connectionless data link protocols does not define a quality of service concept. As such, DLPI does not require every DLS provider implementation to support QOS parameter selection. The DLS provider may specify that any or all QOS parameters are unsupported. This is indicated to the DLS user in the DL\_INFO\_ACK, where the values in the supported range field (indicated by *dl\_qos\_range\_length* and *dl\_qos\_range\_offset*) and the current QOS field (indicated by *dl\_qos\_length* and *dl\_qos\_offset*) of this primitive are set to DL\_UNKNOWN. If the DLS provider supports no QOS parameters, the QOS length fields in the DL\_INFO\_ACK may be set to zero.

If the DLS provider supports QOS parameter selection, the DL\_INFO\_ACK primitive will specify the supported range of parameter values for transit delay, priority, protection and residual error rate. Default values are also returned in the DL\_INFO\_ACK.

For each DL\_UNITDATA\_REQ or DL\_DATA\_ACK\_REQ, the DLS provider should apply the currently selected QOS parameter values to the transmission. If no values have been selected, the default values should be used.

At any point during data transfer, the DLS user may issue a DL\_UDQOS\_REQ primitive to select new values for the transit delay, priority, and protection parameters. These values are selected using the *dl\_qos\_cl\_sel\_t* structure. The residual error rate parameter is ignored by this primitive and cannot be set by a DLS user.

In the DL\_UDQOS\_REQ, the DLS user need not require a specific value for every QOS parameter. DL\_QOS\_DONT\_CARE may be specified if the DLS user does not care what quality of service is provided for a particular parameter. When specified, the DLS provider should retain the current (or default if no previous selection has occurred) value for that parameter.

## *Primitives for Management Services*

This appendix presents the optional primitives to perform essential management functions. The management functions supported are **get** and **set** of physical address, and statistics gathering.

**NAME**

DL\_PHYS\_ADDR\_REQ (*dl\_phys\_addr\_req\_t*) — this primitive requests the DLS provider to return either the default (factory) or the current value of the physical address associated with the stream depending upon the value of the address type selected in the request.

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below:

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_PHYS\_ADDR\_REQ.

*dl\_addr\_type*

conveys the type of address requested — factory physical address or current physical address:

DL\_FACT\_PHYS\_ADDR

factory physical address

DL\_CURR\_PHYS\_ADDR

current physical address.

**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.

**ERRORS**

DL\_NOTSUPPORTED

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

DL\_OUTSTATE

The primitive was issued from an invalid state.

**NAME**

DL\_PHYS\_ADDR\_ACK (*dl\_phys\_addr\_ack\_t*) — this primitive returns the value for the physical address to the link user in response to a DL\_PHYS\_ADDR\_REQ.

**SYNOPSIS**

The message consists of M\_PCPROTO message block containing the following structure:

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_PHYS\_ADDR\_ACK.

*dl\_addr\_length*

conveys length of the physical address.

*dl\_addr\_offset*

conveys the offset from the beginning of the M\_PCPROTO 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.

**NAME**

DL\_SET\_PHYS\_ADDR\_REQ (*dl\_set\_phys\_addr\_req\_t*) — sets the physical address value for all streams for that provider for a particular PPA.

**SYNOPSIS**

The message consists of M\_PROTO message block which contains the following structure:

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

**PARAMETERS**

*dl\_primitive*

conveys DL\_SET\_PHYS\_ADDR\_REQ.

*dl\_addr\_offset*

conveys the offset from the beginning of the M\_PROTO message block.

*dl\_addr\_length*

conveys the length of the requested hardware address.

**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.

**ERRORS**

DL\_BADADDR

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

DL\_BUSY

One or more streams for that particular PPA are in the DL\_BOUND state.

DL\_NOTSUPPORTED

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

DL\_OUTSTATE

The primitive was issued from an invalid state.

DL\_SYSERR

A system error has occurred

**NAME**

DL\_GET\_STATISTICS\_REQ (*dl\_get\_statistics\_req\_t*) — directs the DLS provider to return statistics

**SYNOPSIS**

The message consists of one M\_PROTO message block containing the structure shown below:

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

**PARAMETERS**

*dl\_primitive*  
conveys DL\_GET\_STATISTICS\_REQ.

**STATE**

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

**NEW STATE**

The resulting state is unchanged

**RESPONSE**

The DLS Provider responds to this request with a DL\_GET\_STATISTICS\_ACK if the primitive is supported. Otherwise, a DL\_ERROR\_ACK is returned.

**ERRORS**

DL\_NOTSUPPORTED  
Primitive is known but not supported by the DLS Provider.

## NAME

DL\_GET\_STATISTICS\_ACK (*dl\_get\_statistics\_ack\_t*) — returns statistics in response to the DL\_GET\_STATISTICS\_REQ. The contents of the statistics block is defined in the DLS Provider specific addendum.

## SYNOPSIS

The message consists of one M\_PCPROTO message block containing the structure shown below:

```
typedef struct {
    t_uscalar_t dl_primitive;
    t_uscalar_t dl_stat_length;
    t_uscalar_t offset;
} dl_get_statistics_ack_t;
```

## PARAMETERS

*dl\_primitive*  
conveys DL\_GET\_STATISTICS\_ACK.

*dl\_stat\_len*  
conveys the length of the statistics structure.

*dl\_stat\_offset*  
conveys the offset from the beginning of the M\_PCROTO message block where the statistics information resides.

## STATE

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

## NEW STATE

The resulting state is unchanged

## *Allowable Sequence of DLPI Primitives*

This appendix presents the allowable sequence of DLPI primitives. The sequence is described using a state transition table that defines possible states as viewed by the DLS user.

The state transition table describes transitions based on the current state of the interface and a given DLPI event. Each transition consists of a state change and possibly an interface action.

The states, events, and related transition actions are described below, followed by the state transition table itself.

## B.1 DLPI States

The following table 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 B-1** States Associated with DLPI

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 acknowledgement of a DL_ATTACH_REQ	ALL
2) DETACH PEND	DL_DETACH_PENDING	The DLS user is waiting for an acknowledgement 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 acknowledgement of a DL_BIND_REQ	ALL
5) UNBIND PEND	DL_UNBIND_PENDING	The DLS user is waiting for an acknowledgement 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
7) UDQOS PEND	DL_UDQOS_PENDING	The DLS user is waiting for an acknowledgement 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

State	DLPI State	Description	Service Type
10) CONN_RES PEND	DL_CONN_RES_PENDING	The DLS user is waiting for an acknowledgement of a DL_CONNECT_RES	DL_CODLS
11) DATA XFER	DL_DATA XFER	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 acknowledgement of a DL_RESET_RES	DL_CODLS
15) DISCON 8 PEND	DL_DISCON8_PENDING	The DLS user is waiting for an acknowledgement of a DL_DISCONNECT_REQ issued from the DL_OUTCON_PENDING state	DL_CODLS
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 acknowledgement of a DL_DISCONNECT_REQ issued from the DL_DATA XFER state	DL_CODLS
18) DISCON 12 PEND	DL_DISCON12_PENDING	The DLS user is waiting for an acknowledgement 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 acknowledgement of a DL_DISCONNECT_REQ issued from the DL_PROV_RESET_PENDING state	DL_CODLS

State	DLPI State	Description	Service Type
20) SUBS_BIND PEND	DL_SUBS_BIND_PND	The DLS user is waiting for an acknowledgement of a DL_SUBS_BIND_REQ	ALL
21) SUBS_UNBIND PEND	DL_SUBS_UNBIND_PND	The DLS user is waiting for an acknowledgement of a DL_SUBS_UNBIND_REQ	ALL

## B.2 Variables and Actions for State Transition Table

The following tables describe variables and actions used to describe the DLPI state transitions.

The variables are used to distinguish various uses of the same DLPI primitive. For example, a DL\_CONNECT\_RES causes a different state transition depending on the current number of outstanding connect indications. To distinguish these different connect response events, a variable is used to track the number of outstanding connect indications.

VARIABLE	DESCRIPTION
token	The token contained in a DL_CONNECT_RES that indicates on which stream the connection will be established. A value of zero indicates that the connection will be established on the stream where the DL_CONNECT_IND arrived. A non-zero value indicates the connection will be passed to another stream.
outcnt	Number of outstanding connect indications - those to which the DLS user has not responded. Actions in the state tables that manipulate this value may be disregarded when providing connectionless service.

**Table B-2** DPLI State Transition Table Variables

The actions represent steps the DLS provider must take during certain state transitions to maintain the interface state. When an action is indicated in the state transition table, the DLS provider should change the state as indicated and perform the specified action.

ACTION	DESCRIPTION
1	outcnt = outcnt + 1
2	outcnt = outcnt - 1
3	Pass connection to the stream indicated by the token in the DL_CONNECT_RES primitive

**Table B-3** DLPI State Transition Actions

### B.3 DLPI User-originated Events

The following table describes events initiated by the DLS user that correspond to the various request and response primitives of DLPI. The table presents the event name used in the state transition table, a brief description of the event (including the corresponding DLPI primitive), and an indication of whether the event is valid for connection-oriented data link service (DL\_CODLS), connectionless data link service (DL\_CLDLS), acknowledged connectionless data link service (DL\_ACLDLS) or all.

FSM EVENT	DESCRIPTION	SERVICE TYPE
ATTACH_REQ	DL_ATTACH_REQ primitive	ALL
DETACH_REQ	DL_DETACH_REQ primitive	ALL
BIND_REQ	DL_BIND_REQ primitive	ALL
SUBS_BIND_REQ	DL_SUBS_BIND_REQ primitive	ALL
UNBIND_REQ	DL_UNBIND_REQ primitive	ALL
SUBS_UNBIND_REQ	DL_SUBS_UNBIND_REQ primitive	ALL
UNITDATA_REQ	DL_UNITDATA_REQ primitive	DL_CLDLS
UDQOS_REQ	DL_UDQOS_REQ primitive	DL_CLDLS
CONNECT_REQ	DL_CONNECT_REQ primitive	DL_CODLS
CONNECT_RES	DL_CONNECT_RES primitive	DL_CODLS
PASS_CONN	Received a passed connection from a DL_CONNECT_RES primitive	DL_CODLS
DISCON_REQ	DL_DISCONNECT_REQ primitive	DL_CODLS
DATA_REQ	DL_DATA_REQ primitive	DL_CODLS
RESET_REQ	DL_RESET_REQ primitive	DL_CODLS
RESET_RES	DL_RESET_RES primitive	DL_CODLS
DATA_ACK_REQ	DL_DATA_ACK_REQ primitive	DL_ACLDLS
REPLY_REQ	DL_REPLY_REQ primitive	DL_ACLDLS
REPLY_UPDATE_REQ	DL_REPLY_UPDATE_REQ primitive	DL_ACLDLS

**Table B-4** DLPI User-oriented Events

## **B.4 DLPI Provider-Originated Events**

The following table describes the events initiated by the DLS provider that correspond to the various indication, confirmation, and acknowledgement primitives of DLPI. The table presents the event name used in the state transition table, a brief description of the event (including the corresponding DLPI primitive), and an indication of whether the event is valid for connection-oriented data link service (DL\_CODLS), connectionless data link service (DL\_CLDLS), acknowledged connectionless service (DL\_ACDLS) or all.

FSM EVENT	DESCRIPTION	SERVICE TYPE
BIND_ACK	DL_BIND_ACK primitive	ALL
SUBS_BIND_ACK	DL_SUBS_BIND_ACK primitive	ALL
UNITDATA_IND	DL_UNITDATA_IND primitive	DL_CLDLS
UDERROR_IND	DL_UDERROR_IND primitive	DL_CLDLS
CONNECT_IND	DL_CONNECT_IND primitive	DL_CODLS
CONNECT_CON	DL_CONNECT_CON primitive	DL_CODLS
DISCON_IND1	DL_DISCONNECT_IND primitive when outcnt == 0	DL_CODLS
DISCON_IND2	DL_DISCONNECT_IND primitive when outcnt == 1	DL_CODLS
DISCON_IND3	DL_DISCONNECT_IND primitive when outcnt >1	DL_CODLS
DATA_IND	DL_DATA_IND primitive	DL_CODLS
RESET_IND	DL_RESET_IND primitive	DL_CODLS
RESET_CON	DL_RESET_CON primitive	DL_CODLS
OK_ACK1	DL_OK_ACK primitive when outcnt == 0	ALL
OK_ACK2	DL_OK_ACK primitive when outcnt == 1 and token == 0	DL_CODLS
OK_ACK3	DL_OK_ACK primitive when outcnt == 1 and token != 0	DL_CODLS
OK_ACK4	DL_OK_ACK primitive when outcnt >1 and token != 0	DL_CODLS
ERROR_ACK	DL_ERROR_ACK	ALL
DATA_ACK_IND	DL_DATA_ACK_IND	ACL DLS
DATA_ACK_STATUS_IND	DL_DATA_ACK_STATUS_IND	ACL DLS
REPLY_IND	DL_REPLY_IND	ACL DLS
REPLY_STATUS_IND	DL_REPLY_STATUS_IND	ACL DLS
REPLY_UPDATE_STATUS_IND	DL_REPLY_UPDATE_STATUS_IND	ACL DLS

Table B-5 DLPI Provider-originated Events

## B.5 DLPI State Transition Table

Table B-6, Table B-7, Table B-8, Table B-9 and Table B-10 describe the DLPI state transitions. Each column represents a state of DLPI (see Table B-1 on page 132, and each row represents a DLPI event (see Table B-4 on page 136 and Table B-5 on page 138). The intersecting transition cell defines the resulting state transition (that is, next state) and associated actions, if any, that must be executed by the DLS provider to maintain the interface state. Each cell may contain the following:

- this transition cannot occur.
- n the current input results in a transition to state “n”
- n [a] the list of actions “a” should be executed following the specified state transition “n” (see Table B-3 on page 135 for actions).

The DL\_INFO\_REQ, DL\_INFO\_ACK, DL\_TOKEN\_REQ, and DL\_TOKEN\_ACK primitives are excluded from the state transition table because they can be issued from many states and, when fully processed, do not cause a state transition to occur. However, the DLS user may not issue a DL\_INFO\_REQ or DL\_TOKEN\_REQ if any local acknowledgements are pending. In other words, these two primitives may not be issued until the DLS user receives the acknowledgement for any previously issued primitive that is expecting local positive acknowledgement. Thus, these primitives may not be issued from the following states:

```
DL_ATTACH_PENDING
DL_DETACH_PENDING
DL_BIND_PENDING
DL_SUBS_BIND_PND
DL_SUBS_UNBIND_PND
DL_UNBIND_PENDING
DL_UDQOS_PENDING
DL_CONN_RES_PENDING
DL_RESET_RES_PENDING
DL_DISCON8_PENDING
DL_DISCON9_PENDING
DL_DISCON11_PENDING
DL_DISCON12_PENDING
DL_DISCON13_PENDING
```

Failure to comply by this restriction may result in loss of primitives at the stream head if the DLS user is a user process.

Once a DL\_INFO\_REQ or DL\_TOKEN\_REQ has been issued, the DLS provider must respond with the appropriate acknowledgement primitive.

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 (i.e. would correspond to a "-" cell entry in the state transition table below).
- 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 (see Section 3.2.2 on page 59). 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. This state transition is indicated by the PASS\_CONN event in table 9.
- 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.
- ADL\_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.
- ADL\_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.

The following table presents the allowed sequence of DLPI primitives for the common local management phase of communication.

STATES	UNATT.	ATTACH PEND	DETACH PEND	UNBND	BND PND	UNBND PND	IDLE	SUBS_BND PND	SUBS UNBND PND
EVENTS	0	1	2	3	4	5	6	20	21
ATTACH_REQ	1	-	-	-	-	-	-	-	-
DETACH_REQ	-	-	-	2	-	-	-	-	-
BIND_REQ	-	-	-	4	-	-	-	-	-
BIND_ACK	-	-	-	-	6	-	-	-	-
SUBS_BIND_REQ	-	-	-	-	-	-	20	-	-
SUBS_BIND_ACK	-	-	-	-	-	-	-	6	-
UNBIND_REQ	-	-	-	-	-	-	5	-	-
OK_ACK1	-	3	0	-	-	3	-	-	6
ERROR_ACK	-	0	3	-	3	6	-	-	-
SUBS_UNBND_RQ	-	-	-	-	-	-	21	-	-

**Table B-6** DLPI State Transition - Local Management Phase

The following table presents the allowed sequence of DLPI primitives for the connectionless data transfer phase.

STATES	IDLE	UDQOS PEND
EVENTS	6	7
UDQOS_REQ	7	-
OK_ACK1	-	6
ERROR_ACK	-	6
UNITDATA_REQ	6	-
UNITDATA_IND	6	-
UDERROR_IND	6	-

**Table B-7** DLPI State Transition - Connectionless-mode Data Transfer Phase

The following table presents the allowed sequence of DLPI primitives for the acknowledged connectionless-mode data transfer phase.

EVENTS	STATES IDLE UDQOS PEND	6	7
UDQOS_REQ		7	-
OK_ACK1		-	6
ERROR_ACK		-	6
DATA_ACK_REQ		6	-
REPLY_REQ		6	-
REPLY_UPDATE_REQ		6	-
DATA_ACK_IND		6	-
REPLY_IND		6	-
DATA_ACK_STATUS_IND		6	-
REPLY_STATUS_IND		6	-
REPLY_UPDATE_STATUS_IND		6	-
ERROR_ACK		6	-

**Table B-8** DLPI State Transition - Acknowledged Connectionless-mode Data Transfer Phase

The following table presents the allowed sequence of DLPI primitives for the connection establishment phase of connection mode service.

EVENTS	STATES IDLE OUTCON INCON CONN_RES DATA- XFER DISCON 8 PEND	6	8	9	10	11	15	16
CONNECT_REQ		8	-	-	-	-	-	-
CONNECT_RES		-	-	10	-	-	-	-
DISCON_REQ		-	15	16	-	-	-	-
PASS_CONN		11	-	-	-	-	-	-
CONNECT_IND		9 [1]	-	9 [1]	-	-	-	-
CONNECT_CON		-	11	-	-	-	-	-
DISCON_IND1 (outcnt == 0)		-	6	-	-	6	-	-
DISCON_IND2 (outcnt == 1)		-	-	6 [2]	-	-	-	-
DISCON_IND3 (outcnt > 1)		-	-	9 [2]	-	-	-	-
OK_ACK1 (outcnt == 0)		-	-	-	-	-	6	-
OK_ACK2 (outcnt == 1, token == 0)		-	-	-	11 [2]	-	-	6 [2]
OK_ACK3 (outcnt == 1, token != 0)		-	-	-	6 [2,3]	-	-	6 [2]
OK_ACK4 (outcnt > 1, token != 0)		-	-	-	9 [2,3]	-	-	9 [2]
ERROR_ACK		-	6	-	9	-	8	9

**Table B-9** DLPI State Transition - Connection Establishment Phase

The following table presents the allowed sequence of DLPI primitives for the connection mode data transfer phase.

STATES	IDLE	DATA-XFER	USER RESET PENDING	PROV RESET PENDING	RESET_RES PENDING	DISCON 11 PENDING	DISCON 12 PENDING	DISCON 13 PENDING
EVENTS	6	11	12	13	14	17	18	19
DISCON_REQ	-	17	18	19	-	-	-	-
DATA_REQ	-	11	-	-	-	-	-	-
RESET_REQ	-	12	-	-	-	-	-	-
RESET_RES	-	-	-	14	-	-	-	-
DISCON_IND1 (outcnt == 0)	-	6	6	6	-	-	-	-
DATA_IND	-	11	-	-	-	-	-	-
RESET_IND	-	13	-	-	-	-	-	-
RESET_CON	-	-	11	-	-	-	-	-
OK_ACK1 (outcnt == 0)	-	-	-	-	11	6	6	6
ERROR_ACK	-	-	11	-	13	11	12	13

**Table B-10** DLPI State Transition - Connection-mode Data Transfer Phase



## *Precedence of DLPI Primitives*

### **C.1 Overview**

This appendix presents the precedence of DLPI primitives relative to one another. Two queues are used to describe DLPI precedence rules. One queue contains DLS user-originated primitives and corresponds to the STREAMS write queue of the DLS provider. The other queue contains DLS provider-originated primitives and corresponds to the STREAMS read queue of the DLS user. The DLS provider is responsible for determining precedence on its write queue and the DLS user is responsible for determining precedence on its read queue as indicated in the precedence tables given in this appendix.

For each precedence table, the rows (labeled “PRIM X”) correspond to primitives that are on the given queue and the columns (labeled “PRIM Y”) correspond to primitives that are about to be placed on that queue. Each pair of primitives (PRIM X, PRIM Y) may be manipulated resulting in:

- Change of order, where the order of a pair of primitives is reversed if, and only if, the second primitive in the pair (PRIM Y) is of a type defined to be able to advance ahead of the first primitive in the pair (PRIM X).
- Deletion, where a primitive (PRIM X) may be deleted if, and only if, the primitive that follows it (PRIM Y) is defined to be destructive with respect to that primitive. Destructive primitives may always be added to the queue. Some primitives may cause both primitives in the pair to be destroyed.

The precedence rules define the allowed manipulations of a pair of DLPI primitives. Whether these actions are performed is the choice of the DLS provider for user-originated primitives and the choice of the DLS user for provider-originated primitives.

## C.2 Write Queue Precedence

The following table presents the precedence rules for DLS user-originated primitives on the DLS provider's STREAMS write queue. It assumes that only non-local primitives (i.e. those that generate protocol data units to a peer DLS user) are queued by the DLS provider.

For connection establishment primitives, this table represents the possible pairs of DLPI primitives when connect indications/responses are single-threaded. For the multi-threading scenario, the following rules apply:

- ADL\_CONNECT\_RES primitive has no precedence over either a DL\_CONNECT\_RES or a DL\_DISCONNECT\_REQ primitive that is associated with another connection correlation number (dl\_correlation), and should therefore be placed on the queue behind such primitives.
- Similarly, a DL\_DISCONNECT\_REQ primitive has no precedence over either a DL\_CONNECT\_RES or a DL\_DISCONNECT\_REQ primitive that is associated with another connection correlation number, and should therefore be placed on the queue behind such primitives. Notice, however, that a DL\_DISCONNECT\_REQ does have precedence over a DL\_CONNECT\_RES primitive that is associated with the same correlation number (this is indicated in the table below).

PRIM X (on queue)	PRIM Y	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
P1 DL_INFO_REQ																
P2 DL_ATTACH_REQ																
P3 DL_DETACH_REQ																
P4 DL_BIND_REQ																
P5 DL_UNBIND_REQ																
P6 DL_UNITDATA_REQ							1									
P7 DL_UDQOS_REQ																
P8 DL_CONNECT_REQ												4				
P9 DL_CONNECT_RES												3	1	1		
P10 DL_TOKEN_REQ																
P11 DL_DISCONNECT_REQ									1							
P12 DL_DATA_REQ												5	1	3	3	
P13 DL_RESET_REQ												3				
P14 DL_RESET_RES												3	1	1		
P15 DL_SUBS_BIND_REQ																

**Table C-1** Write Queue Precedence

**KEY:—**

**Code**

"blank"	Empty box indicates a scenario which cannot take place.
1	Y has no precedence over X and should be placed on queue behind X.
2	Y has precedence over X and may advance ahead of X.
3	Y has precedence over X and X must be removed.
4	Y has precedence over X and both X and Y must be removed.
5	Y may have precedence over X (DLS provider's choice) — if so X must be removed.

### C.3 Read Queue Precedence

The following table presents the precedence rules for DLS provider-originated primitives on the DLS user's STREAMS read queue.

For connection establishment primitives, this table represents the possible pairs of DLPI primitives when connect indications/responses are single-threaded. For the multi-threading scenario, the following rules apply:

1. A DL\_CONNECT\_IND primitive has no precedence over either a DL\_CONNECT\_IND or a DL\_DISCONNECT\_IND primitive that is associated with another connection correlation number (dl\_correlation), and should therefore be placed on the queue behind such primitives.
2. Similarly, a DL\_DISCONNECT\_IND primitive has no precedence over either a DL\_CONNECT\_IND or a DL\_DISCONNECT\_IND primitive that is associated with another connection correlation number, and should therefore be placed on the queue behind such primitives.
3. A DL\_DISCONNECT\_IND does have precedence over a DL\_CONNECT\_IND primitive that is associated with the same correlation number (this is indicated in the table below). If a DL\_DISCONNECT\_IND is about to be placed on the DLS user's read queue, the user should scan the read queue for a possible DL\_CONNECT\_IND primitive with a matching correlation number. If a match is found, both the DL\_DISCONNECT\_IND and matching DL\_CONNECT\_IND should be removed.

If the DLS user is a user-level process, its read queue is the stream head read queue. Because a user process has no control over the placement of DLS primitives on the stream head read queue, a DLS user cannot straightforwardly initiate the actions specified in the following precedence table. Except for the connection establishment scenario, the DLS user can ignore the precedence rules defined in the table below. This is equivalent to saying the DLS user's read queue contains at most one primitive.

The only exception to this rule is the processing of connect indication/response primitives. A problem arises if a user issues a DL\_CONNECT\_RES primitive when a DL\_DISCONNECT\_IND is on the stream head read queue. The DLS provider will not be expecting the connect response because it has forwarded the disconnect indication to the DLS user and is in the DL\_IDLE state. It will therefore generate an error upon seeing the DL\_CONNECT\_RES. To avoid this error, the DLS user should not respond to a DL\_CONNECT\_IND primitive if the stream head read queue is not empty. The assumption here is a non-empty queue may be holding a disconnect indication that is associated with the connect indication that is being processed.

When connect indications/responses are single-threaded, a non-empty read queue can only contain a DL\_DISCONNECT\_IND, which must be associated with the outstanding DL\_CONNECT\_IND. This DL\_DISCONNECT\_IND primitive indicates to the DLS user that the DL\_CONNECT\_IND is to be removed. The DLS user should not issue a response to the DL\_CONNECT\_IND if a DL\_DISCONNECT\_IND is received.

The multi-threaded scenario is slightly more complex, because multiple DL\_CONNECT\_IND and DL\_DISCONNECT\_IND primitives may be interspersed on the stream head read queue. In this scenario, the DLS user should retrieve all indications on the queue before responding to a given connect indication.

If a queued primitive is a DL\_CONNECT\_IND, it should be stored by the user process for eventual response. If a queued primitive is a DL\_DISCONNECT\_IND, it should be matched (using the correlation number) against any stored connect indications. The matched connect indication should then be removed, just as is done in the single-threaded scenario.

PRIM X (on queue)	PRIM Y	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
P1 DL_INFO_ACK				1	1	1	1		1	1	1	1			
P2 DL_BIND_ACK				1		1									
P3 DL_UNITDATA_IND	2			1	2								2	2	
P4 DL_UDERROR_IND	2			1	1								2	2	
P5 DL_CONNECT_IND	2							2	4						
P6 DL_CONNECT_CON	2							2	3	1	1				
P7 DL_TOKEN_ACK						1	1		1	1	1	1			
P8 DL_DISCONNECT_IND	2					1		2						2	
P9 DL_DATA_IND	2							2	5	1	3	3		2	
P10 DL_RESET_IND	2							2	3					2	
P11 DL_RESET_CON	2							2	3	1	1			2	
P12 DL_OK_ACK				1	1	1			1	1	1				
P13 DL_ERROR_ACK				1	1	1			1	1	1	1			
P14 DL_SUBS_BIND_ACK				1		1									

Table C-2 Read Queue Precedence

KEY:—

Code

"blank"	Empty box indicates a scenario which cannot take place.
1	Y has no precedence over X and should be placed on queue behind X.
2	Y has precedence over X and may advance ahead of X.
3	Y has precedence over X and X must be removed.
4	Y has precedence over X and both X and Y must be removed.
5	Y may have precedence over X (DLS provider's choice) — if so X must be removed.

## *Guidelines for Protocol-independent DLS Users*

DLPI enables a DLS user to be implemented in a protocol-independent manner such that the DLS user can operate over many DLS providers without changing the DLS user software. DLS user implementors must adhere to the following guidelines, however, to achieve this independence:

- The protocol-specific service limits returned in the DL\_INFO\_ACK primitive (for example, *dl\_max\_sdu*) must not be exceeded. The DLS user should access these limits and adhere to them while interacting with the DLS provider.
- Protocol-specific DLSAP address and PPA identifier formats should be hidden from DLS user software. Hard-coded addresses and identifiers must be avoided. The DLS user should retrieve the necessary information from some other entity (such as a management entity or a higher layer protocol entity) and insert it without inspection into the appropriate primitives.
- The DLS user should not be written to a specific style of DLS provider (that is, style 1 versus style 2). The DL\_INFO\_ACK returns sufficient information to identify which style of provider has been accessed, and the DLS user should perform (or not perform) a DL\_ATTACH\_REQ accordingly.
- The names of devices should not be hard-coded into user-level programs that access a DLS provider.
- The DLS user should access the *dl\_service\_mode* field of the DL\_INFO\_ACK primitive to determine whether connection or connectionless services are available on a given stream.



## DLS Provider-Specific Information

DLPI is a general interface to the services of any DLS provider. However, areas have been documented in this specification where DLS provider-specific information can be conveyed and interpreted. This appendix summarizes all provider-specific issues as an aid to developers of DLS provider implementations. As such, it forms a checklist of required information that should be documented in some manner as part of the provider implementation. The areas DLS provider-specific addendum documentation must address are:

- DLSAP Address Space
- PPA Access and Control
- Quality of Service
- DL\_INFO\_ACK Values
- Supported Services.

For each area listed, a brief description is presented for the associated provider-specific item(s), including references to the appropriate description in this specification.

### E.1 DLSAP Address Space

See reference Section 1.4.2 on page 7, and *DL\_BIND\_REQ* on page 42.

The format of a DLSAP address is specific to each DLS provider, as is the management of that address space. There are no restriction on the format or style of a DLSAP address. As such, a specific implementation should document the format, size, and restrictions of a DLSAP address, as well as information on how the address space is managed. For example, DLPI enables a DLS user to choose a specific DLSAP address to be bound to a stream, but a given implementation may pre-associate addresses with streams based, for example, on the major/minor device number of the stream. In this case, the DLS user could only retrieve the address associated with a stream.

If the DLS provider enables a user to select the DLSAP address for a stream, the implementation must document the contents of the *dl\_sap* field in the *DL\_BIND\_REQ*. This field must contain sufficient information to enable the DLS provider to determine the chosen DLSAP address. This may be the full DLSAP address (if it is not larger than `sizeof ( t_uscalar_t )`), or some distinguishable part of that address. For example, an implementation of a DLS provider conforming to the ISO 8802/2 address space might allow the DSAP or SSAP portion of the DLSAP address to be specified here, where the MAC address portion remains constant over all DLSAP addresses managed by that provider.

Another aspect of address management is whether the provider supports the ability to dynamically allocate DLSAPs other than the requested DLSAP in a *DL\_BIND\_REQ*. Restrictions on DLSAPs might cover the range of supported DLSAP values, services provided by a DLSAP, connection management, and multiplexing. An example of connection management restrictions is the number of connections allowed per DLSAP. Examples of multiplexing restrictions include the number of DLSAPs per PPA, and requirements that certain DLSAPs are attached to specific PPAs.

## E.2 Subsequent DLSAP Addresses

See reference *DL\_SUBS\_BIND\_REQ* on page 49.

The IEEE 802.2 link layer standard allows two ways of specifying a DLSAP value:

- Using an IEEE reserved DLSAP which corresponds to a well-defined protocol.
- Using a privately defined DLSAP.

Previously, subnetworks used privately defined DLSAP values. As these subnetworks move into the OSI world, they may exist in environments with other vendors machines. This presents a problem because there are only 64 privately definable DLSAPS and any other vendor may choose to use these same DLSAP values.

IEEE 802.1 has defined a third way of assigning DLSAP values that will allow for unique private protocol demultiplexing. The *DL\_SUBS\_BIND\_REQ* may be used to support this method.

The subsequent binding of DLSAPs can be peer or hierarchical. When the User requests peer addressing, the *DL\_SUBS\_BIND\_REQ* will specify a DLSAP that may be used in lieu of the DLSAP that was 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 various *ether\_type* values as DLSAPs. The *DL\_BIND\_REQ*, for example, could be issued with *ether\_type* value of *IP*, and a subsequent bind could be issued with *ether\_type* value of *ARP*. The provider may now multiplex off of 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 *DL\_SUBS\_BIND\_REQ* 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 OUI (organizationally unique identifier) 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.

## E.3 PPA Access and Control

See reference Section 1.4.1 on page 6, and Section 3.1.1 on page 31.

A physical point of attachment (PPA) is referenced in DLPI by a PPA identifier, which is of type *t\_uscalar\_t*. The format of this identifier is provider-specific. The DLS provider addendum documentation should describe the format and generation of PPA identifiers for all physical media it is expected to control. It should also describe how a PPA is controlled, the capabilities of the PPA, the number of PPAs supported, and the administrative interface.

Multiplexing capabilities of a PPA should also be described in the DLS provider addendum documentation. This conveys information on the number of DLSAPs that may be supported per PPA, and the number of PPAs supported.

Another item that should be described is the manner in which a PPA is initialized. Section 3.1.1 presents the alternative methods supported by DLPI for initializing a PPA. The interactions of auto-initialization or pre-initialization with the *Attach* and *Bind* services should be discussed, and the following items should be addressed:

- Is auto-initialization, pre-initialization, or both supported for a PPA?
- Can the method of initialization be restricted on a PPA basis?

## E.4 Quality of Service

See reference Chapter 4 on page 111.

Support of QOS parameter negotiation and selection is a provider-specific issue that must be described for each implementation. The DLS provider addendum documentation should describe which, if any, QOS parameters are supported by the provider. For parameters that are negotiated end-to-end, the addendum should describe whether the provider supports end-to-end negotiation, or whether these parameters are negotiated in a local manner only. Finally, default QOS parameter values should be documented.

## E.5 DL\_INFO\_ACK Values

See reference *DL\_INFO\_ACK* on page 34.

The *DL\_INFO\_ACK* primitive specifies information concerning a DLS provider's restrictions and capabilities. The DLS provider addendum documentation should describe the values for all fields in the *DL\_INFO\_ACK*, and how they are determined (static, tunable, dynamic). At a minimum, the addendum must describe the provider style and the service modes supported by the DLS provider.

## E.6 Supported Services

See reference Section 1.2 on page 2.

The overall services that a specific DLS provider supports should be described. These include whether a provider supports connection-mode service, connectionless-mode service (acknowledged or unacknowledged), or both, and how a DLS user selects the appropriate mode. For example, the mode may be mapped directly to a specific major/minor device, and the user selects an appropriate mode by opening the corresponding special file. Alternatively, a DLS provider which supports both modes may enable a DLS user to select the service mode on the *DL\_BIND\_REQ*.

The file name(s) used to access a particular DLS provider and/or specific service modes of that provider must also be documented.



**DLPI Header File**

This appendix contains a listing of the DLPI header file needed by implementations of both DLS user and DLS provider software.

```

#ifndef _SYS_DLPI_H
#define _SYS_DLPI_H
/*
 * dlpi.h header for Data Link Provider Interface
 */

/*
 * This header file has encoded the values so an existing driver
 * or user which was written with the Logical Link Interface ( LLI )
 * can migrate to the DLPI interface in a binary compatible manner.
 * Any fields which require a specific format or value are flagged
 * with a comment containing the message LLI compatibility.
 */

/*
 * DLPI revision definition history
 */
#define DL_CURRENT_VERSION 0x02 /* current version of dlpi */
#define DL_VERSION_2 0x02 /* version of dlpi March 12,1991 */

/*
 * Primitives for Local Management Services
 */
#define DL_INFO_REQ 0x00 /* Information Req, LLI compatibility */
#define DL_INFO_ACK 0x03 /* Information Ack, LLI compatibility */
#define DL_ATTACH_REQ 0x0b /* Attach a PPA */
#define DL_DETACH_REQ 0x0c /* Detach a PPA */
#define DL_BIND_REQ 0x01 /* Bind dlsap address, LLI compatibility */
#define DL_BIND_ACK 0x04 /* Dlsap address bound, LLI compatibility */
#define DL_UNBIND_REQ 0x02 /* Unbind dlsap address, LLI compatibility */
#define DL_OK_ACK 0x06 /* Success acknowledgment, LLI comp. */
#define DL_ERROR_ACK 0x05 /* Error acknowledgment, LLI compatibility */
#define DL_SUBS_BIND_REQ 0x1b /* Bind Subsequent DLSAP address */
#define DL_SUBS_BIND_ACK 0x1c /* Subsequent DLSAP address bound */
#define DL_SUBS_UNBIND_REQ 0x15 /* Subsequent unbind */
#define DL_ENABMULTI_REQ 0x1d /* Enable multicast addresses */
#define DL_DISABMULTI_REQ 0x1e /* Disable multicast addresses */
#define DL_PROMISCON_REQ 0x1f /* Turn on promiscuous mode */
#define DL_PROMISCOFF_REQ 0x20 /* Turn off promiscuous mode */

/*
 * Primitives used for Connectionless Service
 */
#define DL_UNITDATA_REQ 0x07 /* datagram send request, LLI compatibility */
#define DL_UNITDATA_IND 0x08 /* datagram receive indication, LLI comp. */
#define DL_UDERROR_IND 0x09 /* datagram error indication, LLI comp. */
#define DL_UDQOS_REQ 0x0a /* set QOS for subsequent datagram transmits */

/*
 * Primitives used for Connection-Oriented Service

```

```

*/
#define DL_CONNECT_REQ      0x0d /* Connect request */
#define DL_CONNECT_IND      0x0e /* Incoming connect indication */
#define DL_CONNECT_RES      0x0f /* Accept previous connect indication */
#define DL_CONNECT_CON      0x10 /* Connection established */
#define DL_TOKEN_REQ        0x11 /* Passoff token request */
#define DL_TOKEN_ACK        0x12 /* Passoff token ack */
#define DL_DISCONNECT_REQ   0x13 /* Disconnect request */
#define DL_DISCONNECT_IND   0x14 /* Disconnect indication */
#define DL_RESET_REQ        0x17 /* Reset service request */
#define DL_RESET_IND        0x18 /* Incoming reset indication */
#define DL_RESET_RES        0x19 /* Complete reset processing */
#define DL_RESET_CON        0x1a /* Reset processing complete */

/*
 * Primitives used for Acknowledged Connectionless Service
 */
#define DL_DATA_ACK_REQ      0x21 /* data unit transmission request */
#define DL_DATA_ACK_IND      0x22 /* Arrival of a command PDU */
#define DL_DATA_ACK_STATUS_IND 0x23 /* Status indication of DATA_ACK_REQ*/
#define DL_REPLY_REQ        0x24 /* Request a DLSDU from the remote */
#define DL_REPLY_IND        0x25 /* Arrival of a command PDU */
#define DL_REPLY_STATUS_IND 0x26 /* Status indication of REPLY_REQ */
#define DL_REPLY_UPDATE_REQ  0x27 /* Hold a DLSDU for transmission */
#define DL_REPLY_UPDATE_STATUS_IND 0x28 /* Status of REPLY_UPDATE req */

/*
 * Primitives used for XID and TEST operations
 */
#define DL_XID_REQ          0x29 /* Request to send an XID PDU */
#define DL_XID_IND          0x2a /* Arrival of an XID PDU */
#define DL_XID_RES          0x2b /* request to send a response XID PDU*/
#define DL_XID_CON          0x2c /* Arrival of a response XID PDU */
#define DL_TEST_REQ         0x2d /* TEST command request */
#define DL_TEST_IND         0x2e /* TEST response indication */
#define DL_TEST_RES         0x2f /* TEST response */
#define DL_TEST_CON         0x30 /* TEST Confirmation */

/*
 * Primitives to get and set the physical address, and to get Statistics
 */
#define DL_PHYS_ADDR_REQ    0x31 /* Request to get physical addr */
#define DL_PHYS_ADDR_ACK    0x32 /* Return physical addr */
#define DL_SET_PHYS_ADDR_REQ 0x33 /* set physical addr */
#define DL_GET_STATISTICS_REQ 0x34 /* Request to get statistics */
#define DL_GET_STATISTICS_ACK 0x35 /* Return statistics */

/*
 * DLPI interface states
 */
#define DL_UNATTACHED       0x04 /* PPA not attached */
#define DL_ATTACH_PENDING   0x05 /* Waiting ack of DL_ATTACH_REQ */
#define DL_DETACH_PENDING   0x06 /* Waiting ack of DL_DETACH_REQ */
#define DL_UNBOUND          0x00 /* PPA attached, LLI compatibility */
#define DL_BIND_PENDING     0x01 /* Waiting ack of DL_BIND_REQ, */
/* LLI compatibility */
#define DL_UNBIND_PENDING   0x02 /* Waiting ack of DL_UNBIND_REQ, */
/* LLI compatibility */
#define DL_IDLE              0x03 /* dlsap bound, awaiting use, */

```

## DLPI Header File

```

/* LLI compatibility */
#define DL_UDQOS_PENDING      0x07 /* Waiting ack of DL_UDQOS_REQ */
#define DL_OUTCON_PENDING    0x08 /* outgoing connection, */
/* awaiting DL_CONN_CON */
#define DL_INCON_PENDING     0x09 /* incoming connection, */
/* awaiting DL_CONN_RES */
#define DL_CONN_RES_PENDING  0x0a /* Waiting ack of DL_CONNECT_RES */
#define DL_DATAXFER         0x0b /* connection-oriented data transfer */
#define DL_USER_RESET_PENDING 0x0c /* user initiated reset, */
/* awaiting DL_RESET_CON */
#define DL_PROV_RESET_PENDING 0x0d /* provider initiated reset, */
/* awaiting DL_RESET_RES */
#define DL_RESET_RES_PENDING 0x0e /* Waiting ack of DL_RESET_RES */
#define DL_DISCON8_PENDING   0x0f /* Waiting ack of DL_DISC_REQ */
/* when in DL_OUTCON_PENDING */
#define DL_DISCON9_PENDING   0x10 /* Waiting ack of DL_DISC_REQ */
/* when in DL_INCON_PENDING */
#define DL_DISCON11_PENDING  0x11 /* Waiting ack of DL_DISC_REQ */
/* when in DL_DATAXFER */
#define DL_DISCON12_PENDING  0x12 /* Waiting ack of DL_DISC_REQ */
/* when in DL_USER_RESET_PENDING */
#define DL_DISCON13_PENDING  0x13 /* Waiting ack of DL_DISC_REQ */
/* when in DL_DL_PROV_RESET_PENDING */
#define DL_SUBS_BIND_PND     0x14 /* Waiting ack of DL_SUBS_BIND_REQ */
#define DL_SUBS_UNBIND_PND   0x15 /* Waiting ack of DL_SUBS_UNBIND_REQ */

/*
 * DL_ERROR_ACK error return values
 */
#define DL_ACCESS            0x02 /* Improper permissions for request, */
/* LLI compatibility */
#define DL_BADADDR          0x01 /* DLSAP address in improper format or invalid*/
#define DL_BADCORR          0x05 /* Sequence number not from */
/* outstanding DL_CONN_IND */
#define DL_BADDATA          0x06 /* User data exceeded provider limit */
#define DL_BADPPA           0x08 /* Specified PPA was invalid */
#define DL_BADPRIM          0x09 /* Primitive received is not known */
/* by DLS provider */
#define DL_BADQOSPARAM      0x0a /* QOS parameters contained invalid values */
#define DL_BADQOSTYPE       0x0b /* QOS struct. type is unknown or unsupported*/
#define DL_BADSAP           0x00 /* Bad LSAP selector, LLI compatibility */
#define DL_BADTOKEN         0x0c /* Token used not associated with */
/* an active stream */
#define DL_BOUND            0x0d /* Attempted second bind with dl_max_conind */
/* or dl_conn_mgmt >0 on same DLSAP or PPA */
#define DL_INITFAILED       0x0e /* Physical Link initialization failed */
#define DL_NOADDR           0x0f /* Provider couldn't allocate */
/* alternate address */
#define DL_NOTINIT          0x10 /* Physical Link not initialized */
#define DL_OUTSTATE         0x03 /* Primitive issued in improper state, */
/* LLI compatibility */
#define DL_SYSERR           0x04 /* UNIX system error occurred, */
/* LLI compatibility */
#define DL_UNSUPPORTED      0x07 /* Requested service not supplied by provider */
#define DL_UNDELIVERABLE    0x11 /* Previous data unit could not be delivered */
#define DL_NOTSUPPORTED     0x12 /* Primitive is known but not supported */
/* by DLS provider */
#define DL_TOOMANY          0x13 /* limit exceeded */
```

```

#define DL_NOTENAB      0x14 /* Promiscuous mode not enabled */
#define DL_BUSY        0x15 /* Other streams for a particular PPA */
                          /* in the post-attached state */
#define DL_NOAUTO      0x16 /* Automatic handling of XID & TEST */
                          /* responses not supported */
#define DL_NOXIDAUTO   0x17 /* Automatic handling of XID not supported */
#define DL_NOTESTAUTO  0x18 /* Automatic handling of TEST not supported */
#define DL_XIDAUTO     0x19 /* Automatic handling of XID response */
#define DL_TESTAUTO    0x1a /* AUtomatic handling of TEST response*/
#define DL_PENDING     0x1b /* pending outstanding connect indications */

/*
 * NOTE: The range of error codes, 0x80 - 0xff is reserved for
 * implementation specific error codes. This reserved range of error
 * codes will be defined by the DLS Provider.
 */

/*
 * DLPI media types supported
 */
#define DL_CSMACD      0x0  /* IEEE 802.3 CSMA/CD network, LLI Compatibility */
#define DL_TPB         0x1  /* IEEE 802.4 Token Passing Bus, LLI Compatibility */
#define DL_TPR         0x2  /* IEEE 802.5 Token Passing Ring, LLI Compatibility */
#define DL_METRO       0x3  /* IEEE 802.6 Metro Net, LLI Compatibility */
#define DL_ETHER       0x4  /* Ethernet Bus, LLI Compatibility */
#define DL_HDLC        0x05 /* ISO HDLC protocol support, bit synchronous */
#define DL_CHAR        0x06 /* Character Synchronous protocol support, eg BISYNC*/
#define DL_CTCA        0x07 /* IBM Channel-to-Channel Adapter */
#define DL_FDDI        0x08 /* Fiber Distributed data interface */
#define DL_FC          0x10 /* Fibre Channel interface */
#define DL_ATM         0x11 /* ATM */
#define DL_IPATM       0x12 /* ATM Classical IP interface */
#define DL_X25         0x13 /* X.25 LAPB interface */
#define DL_ISDN        0x14 /* ISDN interface */
#define DL_HIPPI       0x15 /* HIPPI interface */
#define DL_100VG       0x16 /* 100 Based VG Ethernet */
#define DL_100VGTPR    0x17 /* 100 Based VG Token Ring */
#define DL_ETH_CSMA    0x18 /* ISO 8802/3 and Ethernet */
#define DL_100BT       0x19 /* 100 Base T */
#define DL_FRAME       0x0a /* Frame Relay LAPF */
#define DL_MPFRAME     0x0b /* Multi-protocol over Frame Relay */
#define DL_ASYNC       0x0c /* Character Asynchronous Protocol */
#define DL_IPX25       0x0d /* X.25 Classical IP interface */
#define DL_LOOP        0x0e /* software loopback */
#define DL_OTHER       0x09 /* Any other medium not listed above */

/*
 * DLPI provider service supported.
 *
 * These must be allowed to be bitwise-OR for dl_service_mode in
 * DL_INFO_ACK.
 */
#define DL_CODLS       0x01 /* support connection-oriented service */
#define DL_CLDLS       0x02 /* support connectionless data link service */
#define DL_ACLDLS      0x04 /* support acknowledged connectionlessservice*/

/*
 * DLPI provider style.
 *

```

## DLPI Header File

```
* The DLPI provider style which determines whether a provider
* requires a DL_ATTACH_REQ to inform the provider which PPA
* user messages should be sent/received on.
*/
#define DL_STYLE1 0x0500 /* PPA is implicitly bound by open(2) */
#define DL_STYLE2 0x0501 /* PPA must be explicitly bound via DL_ATTACH_REQ */

/*
 * DLPI Originator for Disconnect and Resets
 */
#define DL_PROVIDER 0x0700
#define DL_USER 0x0701

/*
 * DLPI Disconnect Reasons
 */
#define DL_CONREJ_DEST_UNKNOWN 0x0800
#define DL_CONREJ_DEST_UNREACH_PERMANENT 0x0801
#define DL_CONREJ_DEST_UNREACH_TRANSIENT 0x0802
#define DL_CONREJ_QOS_UNAVAIL_PERMANENT 0x0803
#define DL_CONREJ_QOS_UNAVAIL_TRANSIENT 0x0804
#define DL_CONREJ_PERMANENT_COND 0x0805
#define DL_CONREJ_TRANSIENT_COND 0x0806
#define DL_DISC_ABNORMAL_CONDITION 0x0807
#define DL_DISC_NORMAL_CONDITION 0x0808
#define DL_DISC_PERMANENT_CONDITION 0x0809
#define DL_DISC_TRANSIENT_CONDITION 0x080a
#define DL_DISC_UNSPECIFIED 0x080b

/*
 * DLPI Reset Reasons
 */
#define DL_RESET_FLOW_CONTROL 0x0900
#define DL_RESET_LINK_ERROR 0x0901
#define DL_RESET_RESYNCH 0x0902

/*
 * DLPI status values for acknowledged connectionless data transfer
 */
#define DL_CMD_MASK 0x0f /* mask for command portion of status */
#define DL_CMD_OK 0x00 /* Command Accepted */
#define DL_CMD_RS 0x01 /* Unimplemented or inactivated service */
#define DL_CMD_UE 0x05 /* Data Link User interface error */
#define DL_CMD_PE 0x06 /* Protocol error */
#define DL_CMD_IP 0x07 /* Permanent implementation dependent error*/
#define DL_CMD_UN 0x09 /* Resources temporarily unavailable */
#define DL_CMD_IT 0x0f /* Temporary implementation dependent error */
#define DL_RSP_MASK 0xf0 /* mask for response portion of status */
#define DL_RSP_OK 0x00 /* Response DLSDU present */
#define DL_RSP_RS 0x10 /* Unimplemented or inactivated service */
#define DL_RSP_NE 0x30 /* Response DLSDU never submitted */
#define DL_RSP_NR 0x40 /* Response DLSDU not requested */
#define DL_RSP_UE 0x50 /* Data Link User interface error */
#define DL_RSP_IP 0x70 /* Permanent implementation dependent error */
#define DL_RSP_UN 0x90 /* Resources temporarily unavailable */
#define DL_RSP_IT 0xf0 /* Temporary implementation dependent error */

/*
 * Service Class values for acknowledged connectionless data transfer
```

```

*/
#define DL_QST_RSP      0x01 /* Use acknowledge capability in MAC sublayer*/
#define DL_QST_NORSP   0x02 /* No acknowledgement service requested */

/*
 * DLPI address type definition
 */
#define DL_FACT_PHYS_ADDR  0x01 /* factory physical address */
#define DL_CURR_PHYS_ADDR  0x02 /* current physical address */

/*
 * DLPI flag definitions
 */
#define DL_POLL_FINAL     0x01 /* if set, indicates poll/final bit set*/

/*
 * XID and TEST responses supported by the provider
 */
#define DL_AUTO_XID       0x01 /* provider will respond to XID */
#define DL_AUTO_TEST      0x02 /* provider will respond to TEST */

/*
 * Subsequent bind type
 */
#define DL_PEER_BIND      0x01 /* subsequent bind on a peer addr */
#define DL_HIERARCHICAL_BIND 0x02 /* subs_bind on a hierarchical addr*/

/*
 * DLPI promiscuous mode definitions
 */
#define DL_PROMISC_PHYS   0x01 /* promiscuous mode at phys level */
#define DL_PROMISC_SAP    0x02 /* promiscuous mode at sap level */
#define DL_PROMISC_MULTI  0x03 /* promiscuous mode for multicast */

/*
 * DLPI Quality Of Service definition for use in QOS structure definitions.
 * The QOS structures are used in connection establishment, DL_INFO_ACK,
 * and setting connectionless QOS values.
 */

/*
 * Throughput
 *
 * This parameter is specified for both directions.
 */
typedef struct {
    t_scalar_t    dl_target_value; /* desired bits/second desired */
    t_scalar_t    dl_accept_value; /* min. acceptable bits/second */
} dl_through_t;

/*
 * transit delay specification
 *
 * This parameter is specified for both directions.
 * expressed in milliseconds assuming a DLSDU size of 128 octets.
 * The scaling of the value to the current DLSDU size is provider dependent.
 */
typedef struct {
    t_scalar_t    dl_target_value; /* desired value of service */

```

## DLPI Header File

```
    t_scalar_t    dl_accept_value; /* min. acceptable value of service */
} dl_transdelay_t;

/*
 * priority specification.
 * priority range is 0-100, with 0 being highest value.
 */
typedef struct {
    t_scalar_t    dl_min;
    t_scalar_t    dl_max;
} dl_priority_t;

/*
 * protection specification
 */
#define DL_NONE      0x0B01 /* no protection supplied */
#define DL_MONITOR   0x0B02 /* protection against passive monitoring */
#define DL_MAXIMUM   0x0B03 /* protection against modification, replay, */
                          /* addition, or deletion */

typedef struct {
    t_scalar_t    dl_min;
    t_scalar_t    dl_max;
} dl_protect_t;

/*
 * Resilience specification
 * probabilities are scaled by a factor of 10,000 with a time interval
 * of 10,000 seconds.
 */
typedef struct {
    t_scalar_t    dl_disc_prob; /* probability of provider init DISC */
    t_scalar_t    dl_reset_prob; /* probability of provider init RESET */
} dl_resilience_t;

/*
 * QOS type definition to be used for negotiation with the
 * remote end of a connection, or aconnectionless unitdata request.
 * There are two type definitions to handle the negotiation
 * process at connection establishment. The typedef dl_qos_range_t
 * is used to present a range for parameters. This is used
 * in the DL_CONNECT_REQ and DL_CONNECT_IND messages. The typedef
 * dl_qos_sel_t is used to select a specific value for the QOS
 * parameters. This is used in the DL_CONNECT_RES,DL_CONNECT_CON,
 * and DL_INFO_ACK messages to define the selected QOS parameters
 * for aconnection.
 *
 * NOTE:
 * A DataLink provider which has unknown values for any of the fields
 * will use a value of DL_UNKNOWN for all values in the fields.
 *
 * NOTE:
 * A QOS parameter value of DL_QOS_DONT_CARE informs the DLS
 * provider the user requesting this value doesn't care
 * what the QOS parameter is set to. This value becomes the
 * least possible value in the range of QOS parameters.
 * The order of the QOS parameter range is then:
 * DL_QOS_DONT_CARE < 0 < MAXIMUM QOS VALUE
 */
#define DL_UNKNOWN      -1
```

```

#define DL_QOS_DONT_CARE      -2

/*
 * Every QOS structure has the first 4 bytes containing a type
 * field, denoting the definition of the rest of the structure.
 * This is used in the same manner has the dl_primitive variable
 * is in messages.
 *
 * The following list is the defined QOS structure type values and structures.
 */
#define DL_QOS_CO_RANGE1     0x0101  /* QOS range structure */
                                   /* for Connection mode service */
#define DL_QOS_CO_SEL1      0x0102  /* QOS selection structure */
#define DL_QOS_CL_RANGE1    0x0103  /* QOS range structure for connectionless*/
#define DL_QOS_CL_SEL1     0x0104  /* QOS selection for connectionless mode*/
typedef struct {
    t_uscalar_t      dl_qos_type;
    dl_through_t     dl_rcv_throughput;    /* desired and acceptable*/
    dl_transdelay_t  dl_rcv_trans_delay;   /* desired and acceptable*/
    dl_through_t     dl_xmt_throughput;
    dl_transdelay_t  dl_xmt_trans_delay;
    dl_priority_t    dl_priority;          /* min and max values */
    dl_protect_t     dl_protection;        /* min and max values */
    t_scalar_t       dl_residual_error;
    dl_resilience_t dl_resilience;
} dl_qos_co_rangel_t;

typedef struct {
    t_uscalar_t      dl_qos_type;
    t_scalar_t       dl_rcv_throughput;
    t_scalar_t       dl_rcv_trans_delay;
    t_scalar_t       dl_xmt_throughput;
    t_scalar_t       dl_xmt_trans_delay;
    t_scalar_t       dl_priority;
    t_scalar_t       dl_protection;
    t_scalar_t       dl_residual_error;
    dl_resilience_t dl_resilience;
} dl_qos_co_sell_t;

typedef struct {
    t_uscalar_t      dl_qos_type;
    dl_transdelay_t  dl_trans_delay;
    dl_priority_t    dl_priority;
    dl_protect_t     dl_protection;
    t_scalar_t       dl_residual_error;
} dl_qos_cl_rangel_t;

typedef struct {
    t_uscalar_t      dl_qos_type;
    t_scalar_t       dl_trans_delay;
    t_scalar_t       dl_priority;
    t_scalar_t       dl_protection;
    t_scalar_t       dl_residual_error;
} dl_qos_cl_sell_t;

/*
 * DLPI interface primitive definitions.
 *
 * Each primitive is sent as a stream message. It is possible that

```

## DLPI Header File

```
* the messages may be viewed as a sequence of bytes that have the
* following form without any padding. The structure definition
* of the following messages may have to change depending on the
* underlying hardware architecture and crossing of a hardware
* boundary with a different hardware architecture.
*
* Fields in the primitives having a name of the form
* dl_reserved cannot be used and have the value of
* binary zero, no bits turned on.
*
* Each message has the name defined followed by the
* stream message type (M_PROTO, M_PCPROTO, M_DATA )
*/

/*
 *          LOCAL MANAGEMENT SERVICE PRIMITIVES
 */

/*
 * DL_INFO_REQ, M_PCPROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* set to DL_INFO_REQ */
} dl_info_req_t;

/*
 * DL_INFO_ACK, M_PCPROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* set to DL_INFO_ACK */
    t_uscalar_t    dl_max_sdu; /* Max bytes in a DLSDU */
    t_uscalar_t    dl_min_sdu; /* Min bytes in a DLSDU */
    t_uscalar_t    dl_addr_length; /* length of DLSAP address */
    t_uscalar_t    dl_mac_type; /* type of medium supported*/
    t_uscalar_t    dl_reserved; /* value set to zero */
    t_uscalar_t    dl_current_state; /* state of DLPI interface */
    t_scalar_t     dl_sap_length; /* current length of SAP part */
    /* of dlsap address */
    t_uscalar_t    dl_service_mode; /* CO, CL or ACL */
    t_uscalar_t    dl_qos_length; /* length of qos values */
    t_uscalar_t    dl_qos_offset; /* offset from beg. of block*/
    t_uscalar_t    dl_qos_range_length; /* available range of qos */
    t_uscalar_t    dl_qos_range_offset; /* offset from beg. of block*/
    t_uscalar_t    dl_provider_style; /* style1 or style2 */
    t_uscalar_t    dl_addr_offset; /* offset of the dlsap addr */
    t_uscalar_t    dl_version; /* version number */
    t_uscalar_t    dl_brdcst_addr_length; /* length of broadcast addr */
    t_uscalar_t    dl_brdcst_addr_offset; /* offset from beg. of block*/
    t_uscalar_t    dl_growth; /* set to zero */
} dl_info_ack_t;

/*
 * DL_ATTACH_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* set to DL_ATTACH_REQ*/
    t_uscalar_t    dl_ppa; /* id of the PPA */
} dl_attach_req_t;
```

```

/*
 * DL_DETACH_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* set to DL_DETACH_REQ */
} dl_detach_req_t;

/*
 * DL_BIND_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* set to DL_BIND_REQ */
    t_uscalar_t    dl_sap; /* info to identify dlsap addr */
    t_uscalar_t    dl_max_conind; /* max # of outstanding con_ind */
    ushort         dl_service_mode; /* CO, CL or ACL */
    ushort         dl_conn_mgmt; /* if non-zero, is con-mgmt stream */
    t_uscalar_t    dl_xidtest_flg; /* if set to 1 indicates automatic */
                                /* initiation of test and xid frames */
} dl_bind_req_t;

/*
 * DL_BIND_ACK, M_PCPROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* DL_BIND_ACK */
    t_uscalar_t    dl_sap; /* DLSAP addr info */
    t_uscalar_t    dl_addr_length; /* length of complete DLSAP addr */
    t_uscalar_t    dl_addr_offset; /* offset from beginning of M_PCPROTO*/
    t_uscalar_t    dl_max_conind; /* allowed max. # of con_ind */
    t_uscalar_t    dl_xidtest_flg; /* responses supported by provider*/
} dl_bind_ack_t;

/*
 * DL_SUBS_BIND_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* DL_SUBS_BIND_REQ */
    t_uscalar_t    dl_subs_sap_offset; /* offset of subs_sap */
    t_uscalar_t    dl_subs_sap_length; /* length of subs_sap */
    t_uscalar_t    dl_subs_bind_class; /* peer or hierarchical */
} dl_subs_bind_req_t;

/*
 * DL_SUBS_BIND_ACK, M_PCPROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* DL_SUBS_BIND_ACK */
    t_uscalar_t    dl_subs_sap_offset; /* offset of subs_sap */
    t_uscalar_t    dl_subs_sap_length; /* length of subs_sap */
} dl_subs_bind_ack_t;

/*
 * DL_UNBIND_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* DL_UNBIND_REQ */
} dl_unbind_req_t;

/*

```

## DLPI Header File

```
* DL_SUBS_UNBIND_REQ, M_PROTO type
*/
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_SUBS_UNBIND_REQ */
    t_uscalar_t    dl_subs_sap_offset;  /* offset of subs_sap */
    t_uscalar_t    dl_subs_sap_length;  /* length of subs_sap */
} dl_subs_unbind_req_t;

/*
 * DL_OK_ACK, M_PCPROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_OK_ACK */
    t_uscalar_t    dl_correct_primitive; /* primitive being acknowledged */
} dl_ok_ack_t;

/*
 * DL_ERROR_ACK, M_PCPROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_ERROR_ACK */
    t_uscalar_t    dl_error_primitive;  /* primitive in error */
    t_uscalar_t    dl_errno;           /* DLPI error code */
    t_uscalar_t    dl_unix_errno;      /* UNIX system error code */
} dl_error_ack_t;

/*
 * DL_ENABMULTI_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_ENABMULTI_REQ */
    t_uscalar_t    dl_addr_length;      /* length of multicast address */
    t_uscalar_t    dl_addr_offset;     /* offset from beg. of M_PROTO block*/
} dl_enabmulti_req_t;

/*
 * DL_DISABMULTI_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_DISABMULTI_REQ */
    t_uscalar_t    dl_addr_length;      /* length of multicast address */
    t_uscalar_t    dl_addr_offset;     /* offset from beg. of M_PROTO block*/
} dl_disabmulti_req_t;

/*
 * DL_PROMISCON_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_PROMISCON_REQ */
    t_uscalar_t    dl_level;           /* physical,SAP level or ALL multicast*/
} dl_promiscon_req_t;

/*
 * DL_PROMISCOFF_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_PROMISCOFF_REQ */
    t_uscalar_t    dl_level;           /* Physical,SAP level or ALL multicast*/
} dl_promiscoff_req_t;
```

```

/*
 * Primitives to get and set the Physical address
 */

/*
 * DL_PHYS_ADDR_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* DL_PHYS_ADDR_REQ */
    t_uscalar_t    dl_addr_type; /* factory or current physical addr */
} dl_phys_addr_req_t;

/*
 * DL_PHYS_ADDR_ACK, M_PCPROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* DL_PHYS_ADDR_ACK */
    t_uscalar_t    dl_addr_length; /* length of the physical addr */
    t_uscalar_t    dl_addr_offset; /* offset from beg. of block */
} dl_phys_addr_ack_t;

/*
 * DL_SET_PHYS_ADDR_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* DL_SET_PHYS_ADDR_REQ */
    t_uscalar_t    dl_addr_length; /* length of physical addr */
    t_uscalar_t    dl_addr_offset; /* offset from beg. of block */
} dl_set_phys_addr_req_t;

/*
 * Primitives to get statistics
 */

/*
 * DL_GET_STATISTICS_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* DL_GET_STATISTICS_REQ */
} dl_get_statistics_req_t;

/*
 * DL_GET_STATISTICS_ACK, M_PCPROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* DL_GET_STATISTICS_ACK */
    t_uscalar_t    dl_stat_length; /* length of statistics structure*/
    t_uscalar_t    dl_stat_offset; /* offset from beg. of block */
} dl_get_statistics_ack_t;

/*
 * CONNECTION-ORIENTED SERVICE PRIMITIVES
 */

/*
 * DL_CONNECT_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive; /* DL_CONNECT_REQ */

```

## DLPI Header File

```
    t_uscalar_t    dl_dest_addr_length; /* len. of dlsap addr */
    t_uscalar_t    dl_dest_addr_offset; /* offset */
    t_uscalar_t    dl_qos_length;      /* len. of QOS parm val */
    t_uscalar_t    dl_qos_offset;      /* offset */
    t_uscalar_t    dl_growth;          /* set to zero */
} dl_connect_req_t;

/*
 * DL_CONNECT_IND, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_CONNECT_IND */
    t_uscalar_t    dl_correlation;      /* provider's correlation token */
    t_uscalar_t    dl_called_addr_length; /* length of called address */
    t_uscalar_t    dl_called_addr_offset; /* offset from beginning of block */
    t_uscalar_t    dl_calling_addr_length; /* length of calling address */
    t_uscalar_t    dl_calling_addr_offset; /* offset from beginning of block */
    t_uscalar_t    dl_qos_length;       /* length of qos structure */
    t_uscalar_t    dl_qos_offset;       /* offset from beginning of block */
    t_uscalar_t    dl_growth;           /* set to zero */
} dl_connect_ind_t;

/*
 * DL_CONNECT_RES, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_CONNECT_RES */
    t_uscalar_t    dl_correlation;      /* provider's correlation token */
    t_uscalar_t    dl_resp_token;       /* token associated with responding stream */
    t_uscalar_t    dl_qos_length;       /* length of qos structure */
    t_uscalar_t    dl_qos_offset;       /* offset from beginning of block */
    t_uscalar_t    dl_growth;           /* set to zero */
} dl_connect_res_t;

/*
 * DL_CONNECT_CON, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_CONNECT_CON */
    t_uscalar_t    dl_resp_addr_length; /* length of responder's address */
    t_uscalar_t    dl_resp_addr_offset; /* offset from beginning of block */
    t_uscalar_t    dl_qos_length;       /* length of qos structure */
    t_uscalar_t    dl_qos_offset;       /* offset from beginning of block */
    t_uscalar_t    dl_growth;           /* set to zero */
} dl_connect_con_t;

/*
 * DL_TOKEN_REQ, M_PCPROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_TOKEN_REQ */
} dl_token_req_t;

/*
 * DL_TOKEN_ACK, M_PCPROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;        /* DL_TOKEN_ACK */
    t_uscalar_t    dl_token;           /* Connection response token */
}
```

```

/* associated with the stream */
} dl_token_ack_t;

/*
 * DL_DISCONNECT_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;    /* DL_DISCONNECT_REQ */
    t_uscalar_t    dl_reason;       /* normal, abnormal, perm. or transient */
    t_uscalar_t    dl_correlation;  /* association with connect_ind */
} dl_disconnect_req_t;

/*
 * DL_DISCONNECT_IND, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;    /* DL_DISCONNECT_IND */
    t_uscalar_t    dl_originator;   /* USER or PROVIDER */
    t_uscalar_t    dl_reason;       /* permanent or transient */
    t_uscalar_t    dl_correlation;  /* association with connect_ind */
} dl_disconnect_ind_t;

/*
 * DL_RESET_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;    /* DL_RESET_REQ */
} dl_reset_req_t;

/*
 * DL_RESET_IND, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;    /* DL_RESET_IND */
    t_uscalar_t    dl_originator;   /* Provider or User */
    t_uscalar_t    dl_reason;       /* flow control, link error or resynch */
} dl_reset_ind_t;

/*
 * DL_RESET_RES, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;    /* DL_RESET_RES */
} dl_reset_res_t;

/*
 * DL_RESET_CON, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;    /* DL_RESET_CON */
} dl_reset_con_t;

/*
 * CONNECTIONLESS SERVICE PRIMITIVES
 */

/*
 * DL_UNITDATA_REQ, M_PROTO type, with M_DATA block(s)
 */

```

## DLPI Header File

```
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_UNITDATA_REQ */
    t_uscalar_t    dl_dest_addr_length;   /* DLSAP length of dest. user */
    t_uscalar_t    dl_dest_addr_offset;   /* offset from beg. of block */
    dl_priority_t  dl_priority;           /* priority value */
} dl_unitdata_req_t;

/*
 * DL_UNITDATA_IND, M_PROTO type, with M_DATA block(s)
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_UNITDATA_IND */
    t_uscalar_t    dl_dest_addr_length;   /* DLSAP length of dest. user */
    t_uscalar_t    dl_dest_addr_offset;   /* offset from beg. of block */
    t_uscalar_t    dl_src_addr_length;    /* DLSAP addr length of sending user */
    t_uscalar_t    dl_src_addr_offset;    /* offset from beg. of block */
    t_uscalar_t    dl_group_address;      /* set to one if multicast/broadcast */
} dl_unitdata_ind_t;

/*
 * DL_UDERROR_IND, M_PROTO type
 * (or M_PCPROTO type if LLI-based provider)
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_UDERROR_IND */
    t_uscalar_t    dl_dest_addr_length;   /* Destination DLSAP */
    t_uscalar_t    dl_dest_addr_offset;   /* Offset from beg. of bloc */
    t_uscalar_t    dl_unix_errno;        /* unix system error code */
    t_uscalar_t    dl_errno;              /* DLPI error code */
} dl_uderror_ind_t;

/*
 * DL_UDQOS_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_UDQOS_REQ */
    t_uscalar_t    dl_qos_length;         /* length in bytes of requested qos */
    t_uscalar_t    dl_qos_offset;         /* offset from beg. of block */
} dl_udqos_req_t;

/*
 * Primitives to handle XID and TEST operations
 */

/*
 * DL_TEST_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_TEST_REQ */
    t_uscalar_t    dl_flag;               /* poll/final */
    t_uscalar_t    dl_dest_addr_length;   /* DLSAP length of dest. user */
    t_uscalar_t    dl_dest_addr_offset;   /* offset from beg. of block */
} dl_test_req_t;

/*
 * DL_TEST_IND, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_TEST_IND */
```

```

    t_uscalar_t    dl_flag;                /* poll/final */
    t_uscalar_t    dl_dest_addr_length;    /* DLSAP length of dest. user */
    t_uscalar_t    dl_dest_addr_offset;    /* offset from beg. of block */
    t_uscalar_t    dl_src_addr_length;     /* DLSAP length of source user */
    t_uscalar_t    dl_src_addr_offset;     /* offset from beg. of block */
} dl_test_ind_t;

/*
 * DL_TEST_RES, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_TEST_RES */
    t_uscalar_t    dl_flag;               /* poll/final */
    t_uscalar_t    dl_dest_addr_length;    /* DLSAP length of dest. user */
    t_uscalar_t    dl_dest_addr_offset;    /* offset from beg. of block */
} dl_test_res_t;

/*
 * DL_TEST_CON, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_TEST_CON */
    t_uscalar_t    dl_flag;               /* poll/final */
    t_uscalar_t    dl_dest_addr_length;    /* DLSAP length of dest. user */
    t_uscalar_t    dl_dest_addr_offset;    /* offset from beg. of block */
    t_uscalar_t    dl_src_addr_length;     /* DLSAP length of source user */
    t_uscalar_t    dl_src_addr_offset;     /* offset from beg. of block */
} dl_test_con_t;

/*
 * DL_XID_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_XID_REQ */
    t_uscalar_t    dl_flag;               /* poll/final */
    t_uscalar_t    dl_dest_addr_length;    /* DLSAP length of dest. user */
    t_uscalar_t    dl_dest_addr_offset;    /* offset from beg. of block */
} dl_xid_req_t;

/*
 * DL_XID_IND, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_XID_IND */
    t_uscalar_t    dl_flag;               /* poll/final */
    t_uscalar_t    dl_dest_addr_length;    /* DLSAP length of dest. user */
    t_uscalar_t    dl_dest_addr_offset;    /* offset from beg. of block */
    t_uscalar_t    dl_src_addr_length;     /* DLSAP length of source user */
    t_uscalar_t    dl_src_addr_offset;     /* offset from beg. of block */
} dl_xid_ind_t;

/*
 * DL_XID_RES, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_XID_RES */
    t_uscalar_t    dl_flag;               /* poll/final */
    t_uscalar_t    dl_dest_addr_length;    /* DLSAP length of dest. user */
    t_uscalar_t    dl_dest_addr_offset;    /* offset from beg. of block */

```

## DLPI Header File

```
} dl_xid_res_t;

/*
 * DL_XID_CON, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_XID_CON */
    t_uscalar_t    dl_flag;               /* poll/final */
    t_uscalar_t    dl_dest_addr_length;   /* DLSAP length of dest. user */
    t_uscalar_t    dl_dest_addr_offset;   /* offset from beg. of block */
    t_uscalar_t    dl_src_addr_length;    /* DLSAP length of source user */
    t_uscalar_t    dl_src_addr_offset;    /* offset from beg. of block */
} dl_xid_con_t;

/*
 * ACKNOWLEDGED CONNECTIONLESS SERVICE PRIMITIVES
 */

/*
 * DL_DATA_ACK_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_DATA_ACK_REQ */
    t_uscalar_t    dl_correlation;        /* User's correlation token */
    t_uscalar_t    dl_dest_addr_length;   /* length of destination addr */
    t_uscalar_t    dl_dest_addr_offset;   /* offset from beginning of block */
    t_uscalar_t    dl_src_addr_length;    /* length of source address */
    t_uscalar_t    dl_src_addr_offset;    /* offset from beginning of block */
    t_uscalar_t    dl_priority;           /* priority */
    t_uscalar_t    dl_service_class;      /* DL_RQST_RSP or DL_RQST_NORS */
} dl_data_ack_req_t;

/*
 * DL_DATA_ACK_IND, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_DATA_ACK_IND */
    t_uscalar_t    dl_dest_addr_length;   /* length of destination addr */
    t_uscalar_t    dl_dest_addr_offset;   /* offset from beginning of block */
    t_uscalar_t    dl_src_addr_length;    /* length of source address */
    t_uscalar_t    dl_src_addr_offset;    /* offset from beginning of block */
    t_uscalar_t    dl_priority;           /* priority for data unit transm. */
    t_uscalar_t    dl_service_class;      /* DL_RQST_RSP or DL_RQST_NORS */
} dl_data_ack_ind_t;

/*
 * DL_DATA_ACK_STATUS_IND, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_DATA_ACK_STATUS_IND */
    t_uscalar_t    dl_correlation;        /* User's correlation token */
    t_uscalar_t    dl_status;             /* success or failure of previous req */
} dl_data_ack_status_ind_t;

/*
 * DL_REPLY_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;           /* DL_REPLY_REQ */
```

```

    t_uscalar_t    dl_correlation;        /* User's correlation token */
    t_uscalar_t    dl_dest_addr_length;   /* length of destination address */
    t_uscalar_t    dl_dest_addr_offset;   /* offset from beginning of block */
    t_uscalar_t    dl_src_addr_length;    /* source address length */
    t_uscalar_t    dl_src_addr_offset;    /* offset from beginning of block */
    t_uscalar_t    dl_priority;           /* priority for data unit transm. */
    t_uscalar_t    dl_service_class;
} dl_reply_req_t;

/*
 * DL_REPLY_IND, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;          /* DL_REPLY_IND */
    t_uscalar_t    dl_dest_addr_length;   /* length of destination address */
    t_uscalar_t    dl_dest_addr_offset;   /* offset from beginning of block */
    t_uscalar_t    dl_src_addr_length;    /* length of source address */
    t_uscalar_t    dl_src_addr_offset;    /* offset from beginning of block */
    t_uscalar_t    dl_priority;           /* priority for data unit transm. */
    t_uscalar_t    dl_service_class;      /* DL_RQST_RSP or DL_RQST_NORS */
} dl_reply_ind_t;

/*
 * DL_REPLY_STATUS_IND, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;          /* DL_REPLY_STATUS_IND */
    t_uscalar_t    dl_correlation;        /* User's correlation token */
    t_uscalar_t    dl_status;             /* success or failure of previous req */
} dl_reply_status_ind_t;

/*
 * DL_REPLY_UPDATE_REQ, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;          /* DL_REPLY_UPDATE_REQ */
    t_uscalar_t    dl_correlation;        /* User's correlation token */
    t_uscalar_t    dl_src_addr_length;    /* length of source address */
    t_uscalar_t    dl_src_addr_offset;    /* offset from beginning of block */
} dl_reply_update_req_t;

/*
 * DL_REPLY_UPDATE_STATUS_IND, M_PROTO type
 */
typedef struct {
    t_uscalar_t    dl_primitive;          /* DL_REPLY_UPDATE_STATUS_IND */
    t_uscalar_t    dl_correlation;        /* User's correlation token */
    t_uscalar_t    dl_status;             /* success or failure of previous req */
} dl_reply_update_status_ind_t;

union DL_primitives {
    t_uscalar_t    dl_primitive;
    dl_info_req_t  info_req;
    dl_info_ack_t  info_ack;
    dl_attach_req_t attach_req;
    dl_detach_req_t detach_req;
    dl_bind_req_t  bind_req;
    dl_bind_ack_t  bind_ack;
    dl_unbind_req_t unbind_req;
}

```

```

dl_subs_bind_req_t      subs_bind_req;
dl_subs_bind_ack_t     subs_bind_ack;
dl_subs_unbind_req_t   subs_unbind_req;
dl_ok_ack_t           ok_ack;
dl_error_ack_t        error_ack;
dl_connect_req_t      connect_req;
dl_connect_ind_t      connect_ind;
dl_connect_res_t      connect_res;
dl_connect_con_t      connect_con;
dl_token_req_t        token_req;
dl_token_ack_t        token_ack;
dl_disconnect_req_t   disconnect_req;
dl_disconnect_ind_t   disconnect_ind;
dl_reset_req_t        reset_req;
dl_reset_ind_t        reset_ind;
dl_reset_res_t        reset_res;
dl_reset_con_t        reset_con;
dl_unitdata_req_t     unitdata_req;
dl_unitdata_ind_t     unitdata_ind;
dl_uderror_ind_t      uderror_ind;
dl_udqos_req_t        udqos_req;
dl_enabmulti_req_t    enabmulti_req;
dl_disabmulti_req_t   disabmulti_req;
dl_promiscon_req_t    promiscon_req;
dl_promiscoff_req_t   promiscoff_req;
dl_phys_addr_req_t    physaddr_req;
dl_phys_addr_ack_t    physaddr_ack;
dl_set_phys_addr_req_t set_physaddr_req;
dl_get_statistics_req_t get_statistics_req;
dl_get_statistics_ack_t get_statistics_ack;
dl_test_req_t         test_req;
dl_test_ind_t         test_ind;
dl_test_res_t         test_res;
dl_test_con_t         test_con;
dl_xid_req_t          xid_req;
dl_xid_ind_t          xid_ind;
dl_xid_res_t          xid_res;
dl_xid_con_t          xid_con;
dl_data_ack_req_t     data_ack_req;
dl_data_ack_ind_t     data_ack_ind;
dl_data_ack_status_ind_t data_ack_status_ind;
dl_reply_req_t        reply_req;
dl_reply_ind_t        reply_ind;
dl_reply_status_ind_t reply_status_ind;
dl_reply_update_req_t reply_update_req;
dl_reply_update_status_ind_t reply_update_status_ind;
};

#define DL_INFO_REQ_SIZE      sizeof(dl_info_req_t)
#define DL_INFO_ACK_SIZE     sizeof(dl_info_ack_t)
#define DL_ATTACH_REQ_SIZE   sizeof(dl_attach_req_t)
#define DL_DETACH_REQ_SIZE   sizeof(dl_detach_req_t)
#define DL_BIND_REQ_SIZE     sizeof(dl_bind_req_t)
#define DL_BIND_ACK_SIZE     sizeof(dl_bind_ack_t)
#define DL_UNBIND_REQ_SIZE   sizeof(dl_unbind_req_t)
#define DL_SUBS_BIND_REQ_SIZE sizeof(dl_subs_bind_req_t)
#define DL_SUBS_BIND_ACK_SIZE sizeof(dl_subs_bind_ack_t)
#define DL_SUBS_UNBIND_REQ_SIZE sizeof(dl_subs_unbind_req_t)
#define DL_OK_ACK_SIZE       sizeof(dl_ok_ack_t)

```

```

#define DL_ERROR_ACK_SIZE          sizeof(dl_error_ack_t)
#define DL_CONNECT_REQ_SIZE       sizeof(dl_connect_req_t)
#define DL_CONNECT_IND_SIZE       sizeof(dl_connect_ind_t)
#define DL_CONNECT_RES_SIZE       sizeof(dl_connect_res_t)
#define DL_CONNECT_CON_SIZE       sizeof(dl_connect_con_t)
#define DL_TOKEN_REQ_SIZE         sizeof(dl_token_req_t)
#define DL_TOKEN_ACK_SIZE         sizeof(dl_token_ack_t)
#define DL_DISCONNECT_REQ_SIZE    sizeof(dl_disconnect_req_t)
#define DL_DISCONNECT_IND_SIZE    sizeof(dl_disconnect_ind_t)
#define DL_RESET_REQ_SIZE         sizeof(dl_reset_req_t)
#define DL_RESET_IND_SIZE         sizeof(dl_reset_ind_t)
#define DL_RESET_RES_SIZE         sizeof(dl_reset_res_t)
#define DL_RESET_CON_SIZE         sizeof(dl_reset_con_t)
#define DL_UNITDATA_REQ_SIZE      sizeof(dl_unitdata_req_t)
#define DL_UNITDATA_IND_SIZE      sizeof(dl_unitdata_ind_t)
#define DL_UDERROR_IND_SIZE       sizeof(dl_uderror_ind_t)
#define DL_UDQOS_REQ_SIZE         sizeof(dl_udqos_req_t)
#define DL_ENABMULTI_REQ_SIZE     sizeof(dl_enabmulti_req_t)
#define DL_DISABMULTI_REQ_SIZE    sizeof(dl_disabmulti_req_t)
#define DL_PROMISCON_REQ_SIZE     sizeof(dl_promiscon_req_t)
#define DL_PROMISCOFF_REQ_SIZE    sizeof(dl_promiscoff_req_t)
#define DL_PHYS_ADDR_REQ_SIZE     sizeof(dl_phys_addr_req_t)
#define DL_PHYS_ADDR_ACK_SIZE     sizeof(dl_phys_addr_ack_t)
#define DL_SET_PHYS_ADDR_REQ_SIZE sizeof(dl_set_phys_addr_req_t)
#define DL_GET_STATISTICS_REQ_SIZE sizeof(dl_get_statistics_req_t)
#define DL_GET_STATISTICS_ACK_SIZE sizeof(dl_get_statistics_ack_t)
#define DL_XID_REQ_SIZE           sizeof(dl_xid_req_t)
#define DL_XID_IND_SIZE           sizeof(dl_xid_ind_t)
#define DL_XID_RES_SIZE           sizeof(dl_xid_res_t)
#define DL_XID_CON_SIZE           sizeof(dl_xid_con_t)
#define DL_TEST_REQ_SIZE          sizeof(dl_test_req_t)
#define DL_TEST_IND_SIZE          sizeof(dl_test_ind_t)
#define DL_TEST_RES_SIZE          sizeof(dl_test_res_t)
#define DL_TEST_CON_SIZE          sizeof(dl_test_con_t)
#define DL_DATA_ACK_REQ_SIZE      sizeof(dl_data_ack_req_t)
#define DL_DATA_ACK_IND_SIZE      sizeof(dl_data_ack_ind_t)
#define DL_DATA_ACK_STATUS_IND_SIZE sizeof(dl_data_ack_status_ind_t)
#define DL_REPLY_REQ_SIZE         sizeof(dl_reply_req_t)
#define DL_REPLY_IND_SIZE         sizeof(dl_reply_ind_t)
#define DL_REPLY_STATUS_IND_SIZE  sizeof(dl_reply_status_ind_t)
#define DL_REPLY_UPDATE_REQ_SIZE  sizeof(dl_reply_update_req_t)
#define DL_REPLY_UPDATE_STATUS_IND_SIZE sizeof(dl_reply_update_status_ind_t)

#endif /* _SYS_DLPI_H */

```

# *Glossary*

**DLPI**

Data Link Provider Interface

**DLS**

Data Link Service

**DLSAP**

Data Link Service Access Point

**DLSDU**

Data Link Service Data Unit

**ISO**

International Organization for Standardization

**OSI**

Open Systems Interconnection

**PPA**

Physical Point of Attachment

**QOS**

Quality of Service

**Called**

The DLS user in connection mode that processes requests for connections from other DLS users.

**Calling**

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 connection 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 management stream**

A special stream that will receive all incoming connect indications destined for DLSAP addresses that are not bound to any other streams associated with a particular PPA.

**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 modes that supports the transfer of data between two DLS users.

**DLSAP**

A point at which a DLS user attaches 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 which accesses the services of the data link layer.

**Local management**

The phase in connection and connectionless modes in which a DLS user initializes 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.

**Quality of service**

Characteristics of transmission quality between two DLS users.

# Index

acknowledged connectionless .....	112	DLS provider .....	176
acknowledged connectionless-mode		DLS provider-specific information .....	151
data transfer service .....	26	DLS user .....	176
service .....	5, 26	DLS users .....	149
service primitives .....	97	DLSAP .....	175-176
address space .....	151	DLSAP address .....	176
allowable sequence of DLPI primitives .....	131	DLSAP address space .....	151
attach service .....	11	DLSDU .....	175
bind service .....	12	DL_ATTACH_REQ .....	39
Bisync .....	1	DL_BIND_ACK .....	46
Called .....	175	DL_BIND_REQ .....	42
Calling .....	175	DL_CONNECT_CON .....	67
Communication endpoint .....	175	DL_CONNECT_IND .....	63
Connection establishment .....	175	DL_CONNECT_REQ .....	61
connection establishment service .....	15	DL_CONNECT_RES .....	65
connection management stream .....	8	DL_DATA_ACK_IND .....	100
Connection management stream .....	175	DL_DATA_ACK_REQ .....	98
Connection mode .....	175	DL_DATA_ACK_STATUS_IND .....	102
Connection release .....	175	DL_DATA_IND .....	71
connection-mode QOS negotiation .....	121	DL_DATA_REQ .....	70
connection-mode sequence flow example .....	28	DL_DETACH_REQ .....	41
connection-mode service .....	4, 15, 111	DL_DISABMULTI_REQ .....	54
connection-mode service primitives .....	59	DL_DISCONNECT_IND .....	74
connectionless .....	112	DL_DISCONNECT_REQ .....	72
Connectionless mode .....	175	DL_ENABMULTI_REQ .....	53
connectionless-mode QOS selection .....	124	DL_ERROR_ACK .....	58
connectionless-mode service .....	5, 23	DL_GET_STATISTICS_ACK .....	130
connectionless-mode service primitives .....	80	DL_GET_STATISTICS_REQ .....	129
CSMA/CD .....	1	DL_INFO_ACK .....	34
Data Link Layer model .....	2	DL_INFO_ACK values .....	153
Data Link Provider Interface .....	1	DL_INFO_REQ .....	33
Data link service data unit .....	175	DL_OK_ACK .....	57
Data Link Service definition .....	1	DL_PHYS_ADDR_ACK .....	127
data link user identification .....	7	DL_PHYS_ADDR_REQ .....	126
Data transfer .....	176	DL_PROMISCOFF_REQ .....	56
data transfer service .....	18	DL_PROMISCON_REQ .....	55
DLPI .....	1, 175	DL_REPLY_IND .....	105
DLPI addressing .....	6	DL_REPLY_REQ .....	103
DLPI header .....	155	DL_REPLY_STATUS_IND .....	106
DLPI primitives .....	29	DL_REPLY_UPDATE_REQ .....	108
DLPI provider-originated events .....	137	DL_REPLY_UPDATE_STATUS_IND .....	109
DLPI services .....	9	DL_RESET_CON .....	79
DLPI state transition table .....	139	DL_RESET_IND .....	77
DLPI states .....	132	DL_RESET_REQ .....	76
DLPI user-originated events .....	136	DL_RESET_RES .....	78
DLS .....	1, 175	DL_SET_PHYS_ADDR_REQ .....	128

DL_SUBS_BIND_ACK .....	51	primitives	
DL_SUBS_BIND_REQ .....	49	acknowledged connectionless-mode .....	97
DL_SUBS_UNBIND_REQ .....	52	allowable .....	131
DL_TEST_CON .....	91	connection-mode .....	59
DL_TEST_IND .....	89	connectionless-mode .....	80
DL_TEST_REQ .....	87	local management .....	31
DL_TEST_RES .....	90	management services .....	125
DL_TOKEN_ACK .....	69	precedence .....	145
DL_TOKEN_REQ .....	68	TEST .....	86
DL_UDERROR_IND .....	84	XID .....	86
DL_UDQOS_REQ .....	85	priority .....	114
DL_UNBIND_REQ .....	48	protection .....	115
DL_UNITDATA_IND .....	83	protocol independence .....	149
DL_UNITDATA_REQ .....	81	provider-specific information .....	151
DL_XID_CON .....	96	address space .....	151
DL_XID_IND .....	94	DL_INFO_ACK values .....	153
DL_XID_REQ .....	92	PPA access and control .....	152
DL_XID_RES .....	95	QOS .....	153
error reporting service .....	24, 27	subsequent DLSAP addresses .....	152
Ethernet .....	1	supported services .....	153
events		QOS .....	111, 153, 175
DLPI provider-originated .....	137	acknowledged connectionless .....	112
DLPI user-originated .....	136	connection-mode negotiation .....	121
FDDI .....	1	connection-mode service .....	111
guidelines for DLS users .....	149	connectionless .....	112
header file .....	155	connectionless-mode selection .....	124
information reporting service .....	11	data structures .....	118
ISDN LAPD .....	1	management service .....	23, 27
ISO .....	175	negotiation .....	121
ISO/IEC 8802-2 .....	1	parameter definitions .....	113
ISO/IEC 8886 .....	1	priority .....	114
LLC .....	1	protection .....	115
Local management .....	176	provider-specific information .....	153
local management service primitives .....	31	residual error rate .....	116
logical link control .....	1	resilience .....	116
management services .....	11	selection .....	121
management services primitives .....	125	throughput .....	113
model of DLL .....	2	transit delay .....	114
model of Service Interface .....	2	quality of service .....	153
modes of communication .....	4	Quality of service .....	176
multi-threaded connection establishment .....	59	read queue precedence .....	147
negotiation .....	121	reset service .....	20
OSI .....	175	residual error rate .....	116
physical attachment identification .....	6	resilience .....	116
PPA .....	175-176	SDLC .....	1
PPA access and control .....	152	selection .....	124
PPA identifier .....	176	Service Interface model .....	2
precedence		state transition table	
read queue .....	147	actions .....	135
write queue .....	146	DLPI .....	139
precedence of DLPI primitives .....	145	variables .....	135

## *Index*

states	
DLPI .....	132
subsequent DLSAP addresses .....	152
supported services .....	153
TEST primitives .....	86
TEST service .....	24
throughput.....	113
token bus .....	1
token ring .....	1
transit delay .....	114
write queue precedence .....	146
X.25 LAPB.....	1
X.25 level 2 .....	1
XID primitives.....	86
XID service .....	24

