Freeway®
Data Link Interface
Reference Guide

DC 900-1385E

Protogate Inc.
12225 World Trade Drive, Suite R
San Diego, CA 92128
March 2002

Protogate, Inc.
12225 World Trade Drive, Suite R
San Diego, CA 92128
(858) 451-0865

Contents

List of Figures 9

List of Tables 11

Preface 13

1 Overview 21

- 1.1 Product Overview . . . 21
  - 1.1.1 Freeway Server . . . 22
  - 1.1.2 Embedded ICP . . . 22
- 1.2 Freeway Client-Server Environment . . . 26
  - 1.2.1 Establishing Freeway Server Internet Addresses . . . 27
- 1.3 Embedded ICP Environment . . . 27
- 1.4 Client Operations . . . 27
  - 1.4.1 Defining the DLI and TSI Configuration . . . 27
  - 1.4.2 Opening a Session . . . 28
  - 1.4.3 Exchanging Data with the Remote Application . . . 28
  - 1.4.4 Closing a Session . . . 28
- 1.5 DLI Overview and Features . . . 28
- 1.6 Protogate's Embedded ICP2432 for the PCIbus . . . 30

2 DLI Concepts 31

- 2.1 Configuration in the Freeway Environment . . . 31
- 2.2 Blocking versus Non-blocking I/O . . . 32
  - 2.2.1 I/O Completion Handler for Non-Blocking I/O . . . 33
- 2.3 Normal versus Raw Operation . . . 34
  - 2.3.1 Normal Operation . . . 35
    - 2.3.1.1 Connecting to the TSI Service Layer . . . 35
    - 2.3.1.2 Connecting to the Message Multiplexor . . . 36

2.3.1.3 Connecting to the ICP 36
2.3.1.4 Configuring the Data Link 36
2.3.1.5 Connecting to the Remote Data Link Application 36
2.3.1.6 Exchanging Data with the Remote Data Link Application . . . 37
2.3.1.7 Disconnecting from the Remote Data Link Application 37
2.3.1.8 Disconnecting from the ICP 37
2.3.1.9 Disconnecting from the Message Multiplexor. 37
2.3.1.10 Disconnecting from the TSI Service Layer 38
2.3.2 Raw Operation 38
2.3.2.1 Connecting to the TSI Service Layer. 39
2.3.2.2 Connecting to the Message Multiplexor. 39
2.3.2.3 Exchanging Data with the Remote Data Link Application . . . 39
2.3.2.4 Disconnecting from the Message Multiplexor. 40
2.3.2.5 Disconnecting from the TSI Service Layer 40
2.4 Buffer Management 40
2.4.1 Overview of the Freeway System Buffer Relationships 40
2.4.1.1 Example Calculation to Change ICP, Client, and Server Buffer Sizes 42
2.4.2 Client TSI Buffer Configuration 45
2.4.2.1 TSI Buffer Pool Definition 46
2.4.2.2 Connection-Specific Buffer Definition 48
2.4.2.3 TSI Buffer Size Negotiation 49
2.4.3 Server TSI Buffer Configuration 50
2.4.4 Buffer Allocation and Release 50
2.4.5 Cautions for Changing Buffer Sizes 51
2.4.6 Using Your Own Buffers 51
2.5 System Resource Requirements 53
2.5.1 Memory Requirements 53
2.5.2 Signal Processing 53

3 DLI Configuration 55

3.1 Configuration Process Overview 55
3.2 DLI Configuration versus TSI Configuration 57
3.3 Introduction to DLI Configuration 58
3.3.1 DLI Configuration Language. 59
3.3.2 Rules of the DLI Configuration File 59

3.3.3 Binary Configuration File Management. 60
3.3.4 On-line Configuration File Processing 61
3.4 DLI Session Definition 61
3.4.1 DLI “main” Configuration Section 62
3.4.2 DLI Session Configuration Sections. 62
3.4.3 Protocol-Specific Parameters for a Session 66
3.5 Miscellaneous DLI Configuration Details 69
3.5.1 DLI Configuration Error Messages 69
3.5.2 Protogate Definition Language (PDL) Grammar. 71

4 DLI Functions 73

4.1 Overview of DLI Functions 73
4.1.1 DLI Error Handling 73
4.1.2 Overview of DLI Functions 74
4.1.2.1 Categories of DLI Functions. 74
4.1.2.2 Summary of DLI Functions 75
4.1.3 DLI Data Structures 78
4.1.3.1 DLI System Configuration. 78
4.1.3.2 DLI Session Status 80
4.1.3.3 DLI Protocol-Specific Optional Arguments 83
4.2 dlBufAlloc 86
4.3 dlBufFree 89
4.4 dlClose 90
4.5 dlControl 95
4.6 dlInit 99
4.7 dlListen 103
4.8 dlOpen 106
4.9 dlpErrString 113
4.10 dlPoll 114
4.11 dlPost 121
4.12 dlRead 122
4.13 dlSyncSelect 128
4.14 dlTerm 132
4.15 dlWrite 134

5 Tutorial Example Programs 141

5.1 Example Program using Blocking I/O 144
5.1.1 DLI Configuration for Blocking I/O and Normal Operation 144
5.1.2 TSI Configuration for Blocking I/O 146
5.1.3 Blocking I/O Example Code Listing 150
5.2 Example Program using Non-Blocking I/O 157
5.2.1 DLI Configuration for Non-Blocking I/O and Normal Operation . . 157
5.2.2 TSI Configuration for Non-Blocking I/O 159
5.2.3 Non-Blocking I/O Example Code Listing 161
5.3 Using Raw Operation 174
5.3.1 Optional Arguments Structure 174
5.4 Example Program using dlControl 179
5.5 Example dlPoll Using usMaxSessBufSize Field 182

A DLI Header Files 185

B DLI Error Codes 187

B.1 Internal Error Codes 187
B.2 Command-Specific Error Codes 195
B.3 Error Handling for Dead Socket Detection 202

C UNIX, VxWorks, and VMS I/O 205

C.1 UNIX Environment 205
C.1.1 Blocking I/O Operations 206
C.1.2 Non-blocking I/O Operations 206
C.1.3 SOLARIS use of SIGALRM 206
C.1.4 Polling I/O Operations 207
C.2 VxWorks Environment 208
C.2.1 Blocking I/O Operations 208
C.2.2 Non-blocking I/O Operations 208
C.3 VMS Environment 209

D DLI Logging and Tracing 211

D.1 DLI Logging 211
D.2 DLI Tracing. 212
D.2.1 Trace Definitions 212

D.2.2 Decoded Trace Layout 214
D.2.3 Example dlidecode Program Output 218
D.2.4 Trace Binary Format 221
D.3 Freeway Server Tracing 222

Index 223

List of Figures

Figure 1–1: Freeway Configuration 23
Figure 1–2: Embedded ICP Configuration 24
Figure 1–3: A Typical Freeway Server Environment 26
Figure 2–1: Client DLI Configuration File Changes (BSC Example) 43
Figure 2–2: Client TSI Configuration File Changes 43
Figure 2–3: Server MuxCfg TSI Configuration File Changes 44
Figure 2–4: TSI Buffer Size Example 47
Figure 2–5: DLI Buffer Size Example 48
Figure 2–6: Comparison of malloc and dlBufAlloc Buffers 52
Figure 2–7: Using the malloc Function for Buffer Allocation 52
Figure 3–1: DLI Overall Architecture 57
Figure 3–2: DLI Example “main” Configuration Section 62
Figure 3–3: DLI Configuration Text File for Two Links 67
Figure 4–1: DLI System Configuration Data Structure 78
Figure 4–2: DLI Session Status Data Structure 80
Figure 4–3: “C” Definition of DLI Optional Arguments Structure 83
Figure 4–4: Freeway DLI Data Format 84
Figure 5–1: Environment for Example Programs 142
Figure 5–2: DLI Text Configuration File for Blocking I/O (fmpssdcfg) 145
Figure 5–3: TSI Text Configuration File for Blocking I/O (fmpsstcfg) 147
Figure 5–4: FMP Blocking I/O Example (fmpssp.c) 154
Figure 5–5: DLI Text Configuration File for Non-Blocking I/O (fmpasdcfg) 158
Figure 5–6: TSI Text Configuration File for Non-Blocking I/O (fmpastcfg) 160
Figure 5–7: FMP Non-Blocking I/O Example (fmpasp.c) 167
Figure 5–8: Link Statistics Report using *Raw* Operation 176

Figure 5–9: Example dlControl Program 179
Figure 5–10: Example dlPoll Program Using usMaxSessBufSize Field 182
Figure D–1: DLI Trace File Format 221
Figure D–2: TRACE_FCB 'C' Structure 221
Figure D–3: DLI_TRACE_HDR "C" Structure 222

List of Tables

Table 2–1: Required Values for Calculating New maxBufSize Parameter 42
Table 3–1: DLI “main” Parameters and Defaults 63
Table 3–2: DLI Client-Related Parameters and Defaults. 64
Table 3–3: DLI Protocol-Specific ICP Link Configuration Parameters 66
Table 4–1: DLI Function Categories 74
Table 4–2: DLI Functions: Syntax and Parameters (Listed in Typical Call Order) . . . 76
Table 4–3: DLI System Configuration Data Structure Fields 79
Table 4–4: DLI Session Status Data Structure Fields. 81
Table 4–5: DLI Protocol-Specific Optional Arguments Data Structure. 84
Table 5–1: TSI “main” Parameters 148
Table 5–2: TSI Connection-Related Parameters 149
Table 5–3: Optional Arguments Required for Raw dlWrite Requests 175
Table A–1: DLI Header Files. 185
Table B–1: DLI Command-specific Error Codes. 195

Preface

Purpose of Document

This document describes Protogate's data link interface (DLI) that helps you develop applications using Protogate's protocol services on a Freeway communications server or embedded intelligent communications processor (ICP).

The information in this document supports the programmer's guide for your particular protocol software running on the Freeway server or embedded ICP. You will need both documents while developing your application.

Note

The DLI information in this document also applies to an embedded ICP using the DLITE interface. If you are using an embedded ICP, also refer to the user guide for your ICP and operating system (for example, the *ICP2432 User Guide for Windows NT*) for more information on the differences between DLI and DLITE.

Intended Audience

This document should be read by application programmers. You should understand the concepts of the client-server interface and be familiar with the C programming language. You should also be familiar with your particular protocol programmer's guide (see the "Protogate References" section below).

Required Equipment

Freeway communications server or embedded ICP.

Organization of Document

Chapter 1 is an overview of Freeway and the DLI.

Chapter 2 describes various DLI concepts that you should understand before writing an application program.

Chapter 3 describes the DLI configuration preprocessor program (dlicfg) and its relationship to the TSI configuration preprocessor program (tsicfg).

Chapter 4 describes each DLI function.

Chapter 5 presents some tutorial example programs demonstrating how to use the DLI functions.

Appendix A describes the DLI header files provided to develop your application.

Appendix B lists additional error codes that are not in the reference sections. It also provides summary tables of all the DLI error codes as they relate to specific DLI functions.

Appendix C compares I/O handling in the UNIX, VMS, and VxWorks environments.

Appendix D describes the DLI logging and tracing capabilities.

Protogate References

The following general product documentation list is to familiarize you with the available Protogate Freeway and embedded ICP products. The applicable product-specific reference documents are mentioned throughout each document (also refer to the “readme” file shipped with each product). Most documents are available on-line at Protogate’s web site, www.protogate.com.

General Product Overviews

• *Freeway 1100 Technical Overview*	25-000-0419
• *Freeway 2000/4000/8800 Technical Overview*	25-000-0374
• *ICP2432 Technical Overview*	25-000-0420
• *ICP6000X Technical Overview*	25-000-0522

Hardware Support

• *Freeway 500 Hardware Installation Guide*	DC-900-2000
• *Freeway 1100/1150 Hardware Installation Guide*	DC-900-1370
• *Freeway 1200/1300 Hardware Installation Guide*	DC-900-1537
• *Freeway 2000/4000 Hardware Installation Guide*	DC-900-1331
• *Freeway 3100 Hardware Installation Guide*	DC-900-2002
• *Freeway 3200 Hardware Installation Guide*	DC-900-2003
• *Freeway 3400 Hardware Installation Guide*	DC-900-2004
• *Freeway 3600 Hardware Installation Guide*	DC-900-2005
• *Freeway 8800 Hardware Installation Guide*	DC-900-1553
• *Freeway ICP6000R/ICP6000X Hardware Description*	DC-900-1020
• *ICP6000(X)/ICP9000(X) Hardware Description and Theory of Operation*	DC-900-0408
• *ICP2424 Hardware Description and Theory of Operation*	DC-900-1328
• *ICP2432 Hardware Description and Theory of Operation*	DC-900-1501
• *ICP2432 Electrical Interfaces (Addendum to DC-900-1501)*	DC-900-1566
• *ICP2432 Hardware Installation Guide*	DC-900-1502

Freeway Software Installation and Configuration Support

• *Freeway Message Switch User Guide*	DC-900-1588
• *Freeway Release Addendum: Client Platforms*	DC-900-1555
• *Freeway User Guide*	DC-900-1333
• *Freeway Loopback Test Procedures*	DC-900-1533

Embedded ICP Software Installation and Programming Support

• *ICP2432 User Guide for Digital UNIX*	DC-900-1513
• *ICP2432 User Guide for OpenVMS Alpha*	DC-900-1511

- *ICP2432 User Guide for OpenVMS Alpha (DLITE Interface)* DC-900-1516
- *ICP2432 User Guide for Solaris STREAMS* DC-900-1512
- *ICP2432 User Guide for Windows NT* DC-900-1510
- *ICP2432 User Guide for Windows NT (DLITE Interface)* DC-900-1514

Application Program Interface (API) Programming Support

- *Freeway Data Link Interface Reference Guide* DC-900-1385
- *Freeway Transport Subsystem Interface Reference Guide* DC-900-1386
- *QIO/SQIO API Reference Guide* DC-900-1355

Socket Interface Programming Support

- *Freeway Client-Server Interface Control Document* DC-900-1303

Toolkit Programming Support

- *Freeway Server-Resident Application and Server Toolkit Programmer Guide* DC-900-1325
- *OS/Impact Programmer Guide* DC-900-1030
- *Protocol Software Toolkit Programmer Guide* DC-900-1338

Protocol Support

- *ADCCP NRM Programmer Guide* DC-900-1317
- *Asynchronous Wire Service (AWS) Programmer Guide* DC-900-1324
- *AUTODIN Programmer Guide* DC-908-1558
- *Bit-Stream Protocol Programmer Guide* DC-900-1574
- *BSC Programmer Guide* DC-900-1340
- *BSCDEMO User Guide* DC-900-1349
- *BSCTRAN Programmer Guide* DC-900-1406
- *DDCMP Programmer Guide* DC-900-1343
- *FMP Programmer Guide* DC-900-1339
- *Military/Government Protocols Programmer Guide* DC-900-1602
- *N/SP-STD-1200B Programmer Guide* DC-908-1359
- *SIO STD-1300 Programmer Guide* DC-908-1559
- *X.25 Call Service API Guide* DC-900-1392
- *X.25/HDLC Configuration Guide* DC-900-1345

- *X.25 Low-Level Interface* DC-900-1307

Document Conventions

This document follows the most significant byte first (MSB) and most significant word first (MSW) conventions for bit-numbering and byte-ordering. In all packet transfers between the client applications and the ICPs, the ordering of the byte stream is preserved.

The term “Freeway” refers to any of the Freeway server models (for example, Freeway 500/3100/3200/3400 PCI-bus servers, Freeway 1000 ISA-bus servers, or Freeway 2000/4000/8800 VME-bus servers). References to “Freeway” also may apply to an embedded ICP product using DLITE (for example, the embedded ICP2432 using DLITE on a Windows NT system).

Physical “ports” on the ICPs are logically referred to as “links.” However, since port and link numbers are usually identical (that is, port 0 is the same as link 0), this document uses the term “link.”

Program code samples are written in the “C” programming language.

File names for the loopback tests and example applications have the format: fmp*xyz…z*

where:	*x*	=	s (blocking I/O)	or	a (non-blocking I/O)
	y	=	l (loopback test)	or	s (sample application)
	z…z	=	p (program)	or	
			dcfg (DLI configuration file)	or	
			tcfg (TSI configuration file)		

Revision History

The revision history of the *Freeway Data Link Interface Reference Guide*, Protogate document DC 900-1385E, is recorded below:

Revision	Release Date	Description
DC 900-1334A	March 1994	Original release
DC 900-1334B (Preliminary)	September 1994	Add Index Reorganize and consolidate Add tutorial examples in Chapter 5
DC 900-1334C	October 1994	Full release
DC 900-1334D	November 1994	Minor corrections and updates throughout Change usICPStatus field to iICPStatus and usProtModifier field to iProtModifier (Table 4–5 on page 84) Update error codes throughout Update Appendix D, “DLI Logging and Tracing”
DC 900-1334E	February 1995	Minor corrections and updated tutorial example programs in Chapter 5.
DC 900-1334F	January 1996	Minor modifications throughout document Add Section 2.5.2 on page 53, “Signal Processing” Add Section 3.3.3 on page 60, “Binary Configuration File Management” Modify Table 3–1 on page 63, Table 3–2 on page 64, and Figure 3–3 on page 67 Add blocking I/O caution on page 115 Add the dlControl function and example code (Section 4.5 on page 95 and Section 5.4 on page 179) Add “QIO/SQIO API User Guide” Appendix Update error codes in Table B–1 on page 195
DC 900-1385A	February 1997	Special version for Freeway Server 2.5 release (changes later incorporated into DC 900-1385B)

Revision	Release Date	Description
DC 900-1385B	October 1997	This is a major revision (using new document number) Transfer “QIO/SQIO API User Guide” Appendix to a separate document, DC-900-1355 Add information for users of Simpact’s embedded ICP2432 PCIbus board (Section 1.6 on page 30) Document DLI_CTRL errors (Section 4.5 on page 95) Add dlpErrString function (Section 4.9 on page 113) Add Freeway Server 2.5 Release modifications (previously released as DC900-1385A), including: Add browser interface for configuration Modify *Buffer Management* Section 2.4 on page 40 Check for NULL value of the piBufLen parameter in the dlPoll request (page 117) Enhance error detection and reporting (Chapter 4 and Appendix B) Add dlPoll example program in Section 5.5 on page 182 Add *Error Handling for Dead Socket Detection* (Section B.3 on page 202)
DC 900-1385C	June 1998	Modify Section 1.1 through Section 1.4 to discuss embedded ICPs Remove browser interface support Add Section 3.3.4 on page 61, “On-line Configuration File Processing” Modify Table 3–1 on page 63 and Table 3–2 on page 64 Add dlSyncSelect function (Section 4.13 on page 128) Add new error codes to Chapter 4 and Appendix B
DC 900-1385D	December 1999	Correct Figure 4–3 on page 83
DC 900-1385E	March 2002	Update contact information for Protogate, Inc. Add references to new Freeway models.

Customer Support

If you are having trouble with any Protogate product, call us at (858) 451-0865 Monday through Friday between 8 a.m. and 5 p.m. Pacific time.

You can also fax your questions to us at (877) 473-0190 any time. Please include a cover sheet addressed to “Customer Service.”

We are always interested in suggestions for improving our products. You can use the report form in the back of this manual to send us your recommendations.

Chapter 1 Overview

This document describes the data link interface (DLI) to Protogate's data link family of protocol services running on Protogate's Freeway communications server. The DLI presents a consistent, high-level, common interface across multiple clients, operating systems, and transport services. The DLI provides session-oriented data services to your application with a subroutine library that implements functions that permit your client application to use data link services to access, configure, establish and terminate sessions, and transfer data across multiple data link protocols.

1.1 Product Overview

Protogate provides a variety of wide-area network (WAN) connectivity solutions for real-time financial, defense, telecommunications, and process-control applications. Protogate's Freeway server offers flexibility and ease of programming using a variety of LAN-based server hardware platforms. Now a consistent and compatible embedded intelligent communications processor (ICP) product offers the same functionality as the Freeway server, allowing individual client computers to connect directly to the WAN.

All models of the Freeway server use the same data link interface (DLI) regardless of the clients' operating system. Also, the embedded ICP uses the DLITE interface which is a subset of DLI. Therefore, migration between the two environments simply requires linking your client application with the proper library. Various client operating systems are supported (for example, Solaris, VMS, and Windows NT).

All Protogate protocols are downloaded to the ICPs and run under the ICP's own CPU and operating system. Programs running on the client operating system interface with the protocols through DLI and TCP/IP (Freeway) or DLITE and an ICP device driver (embedded ICP).

1.1.1 Freeway Server

Protogate's Freeway communications servers enable client applications on a local-area network (LAN) to access specialized WANs through the DLI. The Freeway server can be any of several models (for example, Freeway 3100, Freeway 3200, Freeway 3400, or Freeway 3600). The Freeway server is user programmable and communicates in real time. It provides multiple data links and a variety of network services to LAN-based clients. Figure 1–1 shows the Freeway configuration.

To maintain high data throughput, Freeway uses a multi-processor architecture to support the LAN and WAN services. The LAN interface is managed by a single-board computer, called the server processor. It uses the commercially available VxWorks or BSD UNIX operating system to provide a full-featured base for the LAN interface and layered services needed by Freeway.

Freeway can be configured with multiple WAN interface processor boards, each of which is a Protogate ICP. Each ICP runs the communication protocol software using Protogate's real-time operating system.

1.1.2 Embedded ICP

The embedded ICP connects your client computer directly to the WAN (for example, using Protogate's ICP2432 PCIbus board). The embedded ICP provides client applications with the same WAN connectivity as the Freeway server, using the same data link interface (via the DLITE embedded interface). The ICP runs the communication protocol software using Protogate's real-time operating system. Figure 1–2 shows the embedded ICP configuration.

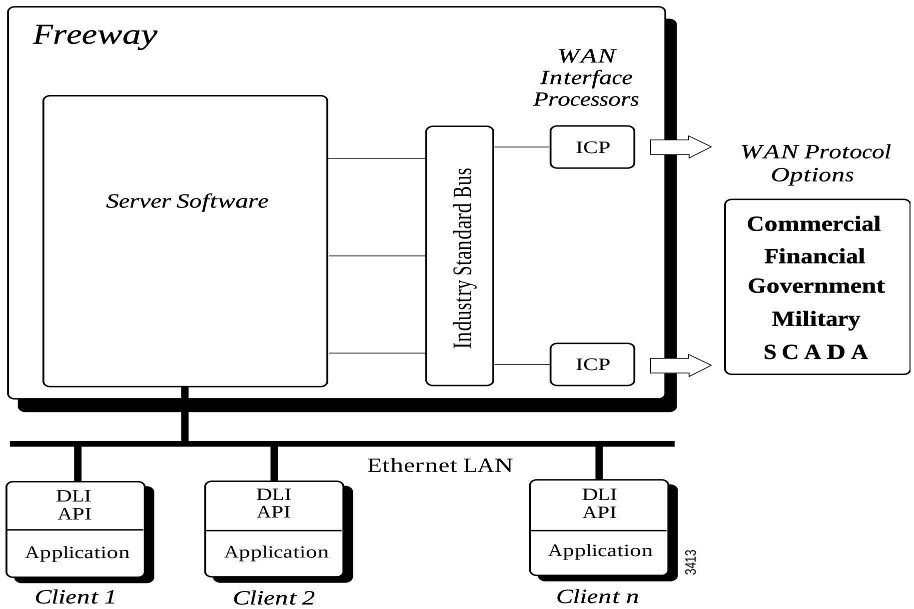

Figure 1–1: Freeway Configuration

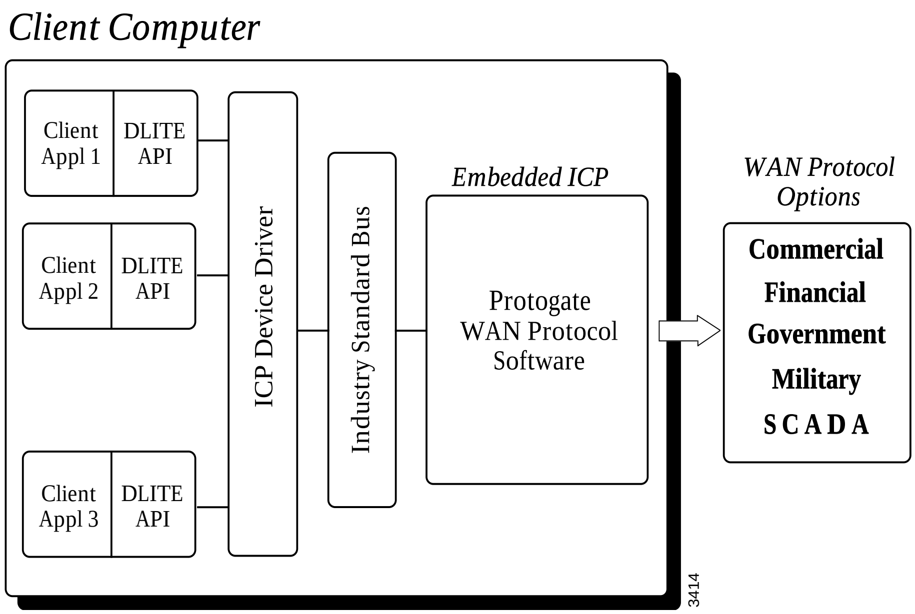

Figure 1–2: Embedded ICP Configuration

Summary of product features:

- Provision of WAN connectivity either through a LAN-based Freeway server or directly using an embedded ICP
- Elimination of difficult LAN and WAN programming and systems integration by providing a powerful and consistent data link interface
- Variety of off-the-shelf communication protocols available from Protogate which are independent of the client operating system and hardware platform
- Support for multiple WAN communication protocols simultaneously
- Support for multiple ICPs (two, four, or eight communication lines per ICP)
- Wide selection of electrical interfaces including EIA-232, EIA-449, EIA-530, and V.35
- Creation of customized server-resident and ICP-resident software, using Protogate's software development toolkits
- Freeway server standard support for Ethernet and Fast Ethernet LANs running the transmission control protocol/internet protocol (TCP/IP)
- Freeway server management and performance monitoring with the simple network management protocol (SNMP), as well as interactive menus available through a local console, telnet, or rlogin

1.2 Freeway Client-Server Environment

The Freeway server acts as a gateway that connects a client on a local-area network to a wide-area network. Through Freeway, a client application can exchange data with a remote data link application. Your client application must interact with the Freeway server and its resident ICPs before exchanging data with the remote data link application.

One of the major Freeway server components is the message multiplexor (MsgMux) that manages the data traffic between the LAN and the WAN environments. The client application typically interacts with the Freeway MsgMux through a TCP/IP BSD-style socket interface (or a shared-memory interface if it is a server-resident application (SRA)). The ICPs interact with the MsgMux through the DMA and/or shared-memory interface of the industry-standard bus to exchange WAN data. From the client application's point of view, these complexities are handled through a simple and consistent data link interface (DLI), which provides dlOpen, dlWrite, dlRead, and dlClose functions.

Figure 1–3 shows a typical Freeway connected to a locally attached client by a TCP/IP network across an Ethernet LAN interface. Running a client application in the Freeway client-server environment requires the basic steps described in Section 1.4.

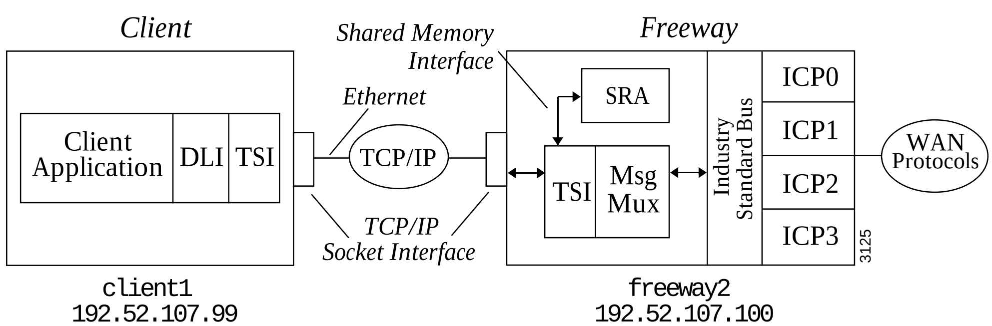

Figure 1–3: A Typical Freeway Server Environment

1.2.1 Establishing Freeway Server Internet Addresses

The Freeway server must be addressable in order for a client application to communicate with it. In the Figure 1–3 example, the TCP/IP Freeway server name is freeway2, and its unique Internet address is 192.52.107.100. The client machine where the client application resides is client1, and its unique Internet address is 192.52.107.99. Refer to the *Freeway User Guide* to initially set up your Freeway and download the operating system, server, and protocol software to Freeway.

1.3 Embedded ICP Environment

Refer to the user guide for your embedded ICP and operating system (for example, the *ICP2432 User Guide for Windows NT*) for software installation and setup instructions. The user guide also gives additional information regarding the data link interface (DLITE) and embedded programming interface descriptions for your specific environment. Refer back to Figure 1–2 on page 24 for a diagram of the embedded ICP environment. Running a client application in the embedded ICP environment requires the basic steps described in Section 1.4, except that DLITE is used instead of DLI and the ICP device driver is used in place of TSI and TCP/IP.

1.4 Client Operations

1.4.1 Defining the DLI and TSI Configuration

You must define the DLI sessions and the transport subsystem interface (TSI) connections between your client application and Freeway (or an embedded ICP). To accomplish this, you first define the configuration parameters in DLI and TSI ASCII configuration files, and then you run two preprocessor programs, dlicfg and tsicfg, to create binary configuration files (see Chapter 3). The dlInit function uses the binary configuration files to initialize the DLI environment.

1.4.2 Opening a Session

After the DLI and TSI configurations are properly defined, your client application uses the dlOpen function to establish a DLI session with an ICP link. As part of the session establishment process, the DLI establishes a TSI connection with the Freeway MsgMux through the TCP/IP BSD-style socket interface for the Freeway server, or directly to the ICP driver for the embedded ICP environment. Section 4.8 on page 106 describes how to use the dlOpen function.

1.4.3 Exchanging Data with the Remote Application

After the link is enabled, the client application can exchange data with the remote application using the dlWrite and dlRead functions. Section 4.12 on page 122 and Section 4.15 on page 134 describe these functions. Also refer to your protocol-specific programmer guide.

1.4.4 Closing a Session

When your application finishes exchanging data with the remote application, it calls the dlClose function to disable the ICP link, close the session with the ICP, and disconnect from Freeway or the embedded ICP driver. Section 4.5 on page 95 describes dlClose.

1.5 DLI Overview and Features

The data link interface (DLI) provides a set of flexible and easy-to-use functions to establish, maintain and terminate a *session* with a remote data link application through Protogate's Freeway communication server. The DLI allows the application programmer to exchange commands and responses with the remote data link application by shielding the application from the details of interacting with the Freeway server and the data link protocols supported by Freeway. Your application might not need all the capabilities that the DLI provides; however, careful system design and consideration will allow your application not only to have a longer useful life but also to be ported across various operating environments.

The DLI requires the underlying Freeway transport subsystem interface (TSI). Similar to the DLI, the TSI provides a set of flexible and easy-to-use functions to establish, maintain, and terminate a *connection* with a remote transport application. For more information on the TSI, refer to the *Freeway Transport Subsystem Interface Reference Guide*.

The DLI's flexible configuration services across different data link protocols are easily portable to various operating environments. For example, DLI can be configured to operate in an environment where both system and network resources are scarce. These configuration services are provided through a free-formatted, procedure-like definition language that is simple to use and yet powerful enough to satisfy your complex data link application requirements.

The DLI provides your application with the choice of *Normal* or *Raw* operation, plus the choice of blocking or non-blocking I/O. These can be chosen in any combination. These concepts are explained in Chapter 2.

The DLI major features are summarized as follows:

- Uses Protogate's TSI services to communicate with the Freeway server
- Provides both protocol-independent (*Normal*) and protocol-dependent (*Raw*) data link operations
- Permits transport-service-independent applications (using the TSI)
- Supports multiple TSI connections to multiple servers
- Supports blocking I/O
- Supports non-blocking I/O, using notification by I/O completion handler (IOCH) or polling

- Provides advanced queuing techniques to minimize internal task switches under the VxWorks operating system
- Provides efficient buffer management to avoid excess memory movement
- Provides flexible text-based configuration services
- Provides off-line configuration preprocessor programs (dlicfg and tsicfg) to increase syntax and semantic checking capability and to reduce real-time (on-line) processing of the configuration parameters
- Provides configuration for most data link protocol-specific parameters
- Provides configuration for all significant DLI service parameters

1.6 Protogate's Embedded ICP2432 for the PCIbus

Protogate's ICP2432 PCIbus board can be installed in a Freeway server (such as the Freeway 3400), or embedded in a PCIbus computer. Programmers writing an application interfacing to the embedded ICP2432 use the DLITE interface which provides access to a particular ICP2432 device driver. Since DLITE is a subset of DLI, the effort to port existing Freeway DLI applications to the embedded ICP environment is minimal.

Windows NT users can use the DLITE through the "NTsi" local interface. For details on using DLITE with NTsi, refer to the *ICP2432 User Guide for Windows NT* in conjunction with this *Freeway Data Link Interface Reference Guide*.

Chapter 2

DLI Concepts

Note

The DLI concepts in this chapter also apply to an embedded ICP using the DLITE interface. If you are using an embedded ICP, also refer to the user guide for your ICP and operating system for concepts specific to DLITE.

The following DLI concepts are described in this chapter:

- configuration at various levels of the Freeway environment
- blocking versus non-blocking I/O
- *Normal* versus *Raw* operation
- buffer management
- system resource requirements

2.1 Configuration in the Freeway Environment

There are several types of configuration required for a client application to run in the Freeway environment:

- Freeway server configuration
- data link interface (DLI) session configuration

- transport subsystem interface (TSI) connection configuration
- protocol-specific ICP link configuration

The Freeway server is normally configured only once, during the installation procedures described in the *Freeway User Guide*. DLI session and TSI connection configurations are defined by specifying parameters in DLI and TSI ASCII configuration files and then running two preprocessor programs, dlicfg and tsicfg, to create binary configuration files. Chapter 5 covers some of the TSI configuration parameters in conjunction with the tutorial programs; see the *Freeway Transport Subsystem Interface Reference Guide* for complete details.

ICP link configuration can be performed using any of the following methods:

- The dlOpen function (Section 4.8 on page 106) can configure the ICP links during the DLI session establishment process using the default ICP link configuration values provided by the protocol software.
- You can specify ICP link parameters in the DLI ASCII configuration file and then run the dlicfg preprocessor program (Chapter 3). The dlOpen function (Section 4.8 on page 106) uses the resulting DLI binary configuration file to perform the link configuration during the DLI session establishment process.
- You can perform ICP link configuration within the client application (refer to your particular protocol programmer's guide). This method is useful if you need to change link configuration without exiting the application.

2.2 Blocking versus Non-blocking I/O

Note

Earlier Freeway releases used the term "synchronous" for blocking I/O and "asynchronous" for non-blocking I/O. Some parameter names reflect the previous terminology.

Non-blocking I/O applications are useful when doing I/O to multiple channels with a single process where it is not possible to "block" (sleep) on any one channel. Blocking I/O applications are useful when it is reasonable to have the calling process wait for I/O completion. For example, if you wish to design an application requiring the input of a keyboard as well as background processing, non-blocking I/O would be more efficient, because your process can perform other tasks while waiting for keyboard input.

In the Freeway environment, the term blocking I/O indicates that the dlOpen, dlClose, dlRead and dlWrite functions do not return until the I/O is complete. For non-blocking I/O, these functions might return after the I/O has been queued at the client, but before the transfer to the Freeway server is complete. The client must handle I/O completions at the software interrupt level in the completion handler established by the dlInit or dlOpen function, or by periodic use of dlPoll to query the I/O completion status.

The asyncIO DLI configuration parameter (page 63) specifies whether an application session uses blocking or non-blocking I/O (set asyncIO to "no" to use blocking I/O, which is the default). The alwaysQIO DLI configuration parameter (page 64) further qualifies the operation of non-blocking I/O activity.

The effects on different DLI functions, resulting from the choice of blocking or non-blocking I/O, are explained in each function description of Chapter 4. The tutorial example programs in Chapter 5 demonstrate the use of blocking and non-blocking I/O.

2.2.1 I/O Completion Handler for Non-Blocking I/O

When your application uses non-blocking I/O and an I/O condition occurs, the current task is preempted by a high-priority task called an I/O completion handler (IOCH) which is designated to handle the I/O. This high-priority IOCH is written by the application programmer and should adhere to the following real-time criteria to prevent the IOCH from impacting overall system performance:

- minimize the amount of processing performed within the IOCH so other vital system operations are not prevented from executing

- allow the non-preemptive priority routines to complete the processing
- avoid activities such as disk I/O which might block the operations

2.3 Normal versus Raw Operation

There are two choices for the protocol DLI configuration parameter (page 65):

- A session is opened for *Normal* operation if you set protocol to a specific protocol (for example, "FMP" or "BSC3780"); then the DLI software configures the ICP links using the values in the DLI configuration file and transparently handles all headers and I/O.

- A session is opened for *Raw* operation if you set protocol to "raw"; then your application must handle all configuration, headers, and I/O details. *Raw* operation is recommended for data transfer where responses might be received out of sequence (for example, when using the BSC 3270 protocol).

Normal and *Raw* operations can be mixed. For example, the client application session can be configured for *Normal* operation (allowing DLI to handle link startup and configuration), but the read and write requests (Section 4.12 on page 122 and Section 4.15 on page 134) can use *Raw* operation by including the optional arguments structure containing the protocol-specific information (Section 4.1.3.3 on page 83).

Note

The protocol-specific writeType DLI configuration parameter (page 66) specifies the type of data to be sent on the line (typically normal or transparent). This parameter should not be confused with *Normal* operation.

The details of *Normal* and *Raw* operation are explained in Section 2.3.1 and Section 2.3.2 to assist you in writing and debugging your DLI applications.

2.3.1 Normal Operation

In *Normal* operation, your application is not concerned with the interactive commands and responses exchanged between your application and the Freeway server or with the details of various supported data link protocols. DLI *Normal* operation uses the following hierarchy:

1. The DLI uses the transport subsystem interface (TSI) component to interact with the locally attached Freeway server.
2. The DLI communicates directly to the message multiplexor (MsgMux) that operates on the Freeway main processor board.
3. Through the MsgMux, the DLI communicates with the ICP and the ICP-resident protocol services.
4. Through the ICP and its protocol service, the DLI exchanges data with the remote data link application.

Normal operation is broken into the steps listed in Section 2.3.1.1 through Section 2.3.1.10. The DLI handles these actions automatically as the application uses the dlOpen, dlRead, dlWrite and dlClose functions during *Normal* operation. Refer back to Figure 1–3 on page 26 for an overview of the DLI operating environment.

2.3.1.1 Connecting to the TSI Service Layer

The dlOpen function connects to the locally attached Freeway using the tConnect function in the TSI service layer. Your session definition must specify to the DLI which TSI connection name your session will use (transport parameter on (page 65). After dlOpen makes the TSI connection, it is ready to communicate with the MsgMux component of the Freeway server.

2.3.1.2 Connecting to the Message Multiplexor

The dlOpen function then sends a DLI_FW_OPEN_SESS_CMD to the Freeway MsgMux. The MsgMux responds with a DLI_FW_OPEN_SESS_RSP control packet if it is available to manage one more session with the DLI. If the MsgMux is able to accept the session request, dlOpen proceeds to connect to the desired ICP.

2.3.1.3 Connecting to the ICP

The dlOpen function then sends a protocol-specific request to start communications with the designated ICP on Freeway. For example, for the FMP and BSC protocols, dlOpen sends a DLI_ICP_CMD_ATTACH request (without a protocol-specific command) to the ICP protocol service. The ICP responds whether or not it is able to honor the request. If this step completes successfully, dlOpen proceeds to configure the data link.

Note

The protocol between the DLI and the ICP-resident protocol services is subject to change in future releases.

2.3.1.4 Configuring the Data Link

If your session definition requires data link configuration prior to its use, dlOpen then sends the protocol-specific configuration request prior to connecting to the remote data link application. After configuring the data link, dlOpen proceeds to connect to the remote data link application.

2.3.1.5 Connecting to the Remote Data Link Application

The dlOpen function then sends a protocol-specific request to connect with the designated remote data link application. For example, for the FMP and BSC protocols, dlOpen sends a DLI_ICP_CMD_BIND request (without a protocol-specific command) to the defined ICP and port. If the session is configured for blocking I/O, dlOpen returns a session ID to your application when it successfully connects to the remote data link application. Data can now be exchanged between the two applications.

2.3.1.6 Exchanging Data with the Remote Data Link Application

After receiving the session ID from dlOpen, your application can now exchange data with the remote data link application using the dlWrite and dlRead requests. However, your application cannot issue commands either to the Freeway server to open or close sessions, or to the ICP protocol server to disconnect from the remote data link application or to detach your application from the ICP. In *Normal* operation, these commands are reserved for the DLI only.

Your application can also exchange data with the Freeway MsgMux component, and the ICP protocol service using *Raw* operation.

2.3.1.7 Disconnecting from the Remote Data Link Application

To terminate a connection between an ICP link and a remote data link application, dlClose sends a protocol-specific command to the ICP. For the FMP and BSC protocols, dlClose sends the DLI_ICP_CMD_UNBIND request.

2.3.1.8 Disconnecting from the ICP

To disconnect from the ICP, the dlClose function then sends a protocol-specific command to the ICP protocol service. For the FMP and BSC protocol, dlClose sends the DLI_ICP_CMD_DETACH request.

2.3.1.9 Disconnecting from the Message Multiplexor

Next, the dlClose function sends the DLI_FW_CMD_CLOSE_SESS_CMD request to the Freeway MsgMux. If the MsgMux accepts the request, dlClose proceeds to disconnect from the TSI service layer.

2.3.1.10 Disconnecting from the TSI Service Layer

The dlClose function issues a tDisconnect request to the TSI service layer to close the TSI connection.

2.3.2 Raw Operation

If your application requires protocol-specific information such as ICP link statistics or link configuration, or performs data transfer requests other than for single packets, it can use *Raw* operation to do these functions. Use of *Raw* operation is recommended whenever your application must interface with the protocol software for any reason outside of simple data transfer.

Recall that to be a DLI fully *Raw* application, your application must open a session with the protocol DLI configuration parameter (page 65) defined as “raw”. In *Raw* operation, your application takes full control of the I/O operations between it, Freeway, and the remote data link application. The DLI manages only the connection between your application and the message multiplexor (MsgMux) subsystem component of the Freeway server. You must fully understand the interactive commands and responses required between your application and the Freeway server, as well as each data link protocol that your application must program.

To perform *Raw* read and write requests, your application must use the optional arguments structure (Section 4.1.3.3 on page 83) to pass all necessary protocol-specific information to the DLI. By providing the optional arguments instead of the actual DLI headers, the DLI extends its portability and minimizes software modification between DLI releases.

Raw operation is similar to *Normal* operation. In *Raw* operation, however, dlOpen completes after it successfully connects to the Freeway MsgMux component. For the closing process, dlClose disconnects from the Freeway MsgMux and then calls tDisconnect to close the TSI connection. Your DLI application must ensure proper disconnection from the remote data link as well as the ICP protocol service. Improper disconnection could cause the Freeway MsgMux, as well as the ICP protocol service, to enter an undefined state.

Raw operation allows the addition of WAN protocols developed with the protocol toolkit product provided by Protogate, described in the *Protocol Software Toolkit Programmer Guide*. *Raw* operation is broken into the steps listed in Section 2.3.2.1 through Section 2.3.2.5. The DLI handles these actions automatically as the application uses the dlOpen, dlRead, dlWrite and dlClose functions during *Raw* operation.

2.3.2.1 Connecting to the TSI Service Layer

The dlOpen function connects to the locally attached Freeway using the tConnect function in the TSI service layer. Your session definition must specify to the DLI which TSI connection name your session will use (transport parameter on page 65). After dlOpen makes the TSI connection, it is ready to communicate with the MsgMux component of the Freeway server.

2.3.2.2 Connecting to the Message Multiplexor

The dlOpen function then sends a DLI_FW_OPEN_SESS_CMD to the Freeway MsgMux. The MsgMux responds with a DLI_FW_OPEN_SESS_RSP control packet if it is available to manage one more session with the DLI. If the MsgMux is able to accept the session request, dlOpen completes its opening process and returns a session ID to your application.

2.3.2.3 Exchanging Data with the Remote Data Link Application

After receiving the session ID, your application can now exchange data with the Freeway MsgMux, the ICP, and subsequently with the remote data link application using the dlWrite and dlRead requests. The DLI will not interfere with your data transfer procedures, except it will not allow commands to the Freeway MsgMux component to open or close sessions. These commands are reserved for the DLI only.

2.3.2.4 Disconnecting from the Message Multiplexor

After completion of data transfer, your application uses the dlClose function to send the DLI_FW_CMD_CLOSE_SESS_CMD request to the Freeway MsgMux. If the MsgMux accepts the request, dlClose proceeds to disconnect from the TSI service layer.

2.3.2.5 Disconnecting from the TSI Service Layer

The dlClose function issues a tDisconnect request to the TSI service layer to close the TSI connection.

2.4 Buffer Management

This section describes how the Freeway buffer management system operates. For users who do not need a detailed understanding of the system design, Section 2.4.1 gives a system buffer overview and an example for reconfiguring your system buffers. Section 2.4.2 through Section 2.4.6 give the detailed information for those interested.

Note

Freeway buffer management is implemented in the TSI; however DLI uses the TSI system for its own buffer management. Therefore, the DLI perspective is also presented throughout this section. If your application interfaces to the TSI only (not the DLI), you can disregard the DLI-specific information.

2.4.1 Overview of the Freeway System Buffer Relationships

In the Freeway environment, user-configurable buffers exist in the ICP, the client, and the server. These buffers must be coordinated for proper operation between the client application, the Freeway server, and the ICP. The default sizes for each of these buffers are designed for operation in a typical Freeway system. However, if your system requires reconfiguration of buffer sizes, the basic procedure is as follows (Section 2.4.1.1 gives an example calculation):

Step 1: As a general rule, define the ICP buffer size first. ICP buffers must be large enough to contain the largest application data buffer transmitted or received. Most Protogate protocols on a Freeway ICP provide a data link interface (DLI) configuration parameter (such as msgBlkSize for BSC) through which the user can configure the ICP message buffer size. The typical default ICP buffer size for most Protogate protocols is 1024. Refer to your protocol-specific *Programmer's Guide* to determine the parameter name and default.

Note

If your application does not interface to the DLI, the protocol-specific ICP buffer size is also software configurable. Refer to your protocol-specific *Programmer's Guide*.

Step 2: Define the client buffers in the client's TSI configuration file. The TSI buffer pool is defined in the configuration file's "main" section. An optional connection-specific maximum buffer size is allowed in each connection definition. These two configurations are detailed in Section 2.4.2.1 and Section 2.4.2.2, respectively. The buffer size specified in the associated connection definition must be large enough to contain the ICP buffer size.

Note

If your application uses the DLI, the client buffer size must also be large enough to contain the DLI header.

Step 3: Define the server buffers in the MuxCfg server TSI configuration file, which is located in your boot directory. This file is similar to the client TSI configuration file. As with the client, define the TSI buffer pool size in the MuxCfg file's "main" section. Then define the optional connection-specific maximum buffer size for each connection. Simply define the connection buffer size for the largest associated client requirement. The buffer pool size must be at least as large as the largest connection buffer size. The

Freeway Transport Subsystem Interface Reference Guide discusses the MuxCfg file in detail.

2.4.1.1 Example Calculation to Change ICP, Client, and Server Buffer Sizes

Step 1: Determine the maximum bytes of data your application must be able to transfer. For this example calculation, we are assuming a maximum of 1500 bytes to be transferred using the BSC protocol and interfacing to Protogate's DLI. This is the value that must be assigned to the ICP buffer size (the DLI msgBlkSize parameter for BSC).

Step 2: Based on the above 1500-byte msgBlkSize parameter, calculate a new maxBufSize for the ICP, client and server. Table 2–1 summarizes the values used in this example.

maxBufSize = msgBlkSize + DLI header size

maxBufSize = 1500 bytes + 96 bytes = 1596 bytes

Table 2–1: Required Values for Calculating New maxBufSize Parameter

Item	Requirement	Description
BSC msgBlkSize parameter[1]	1500 bytes	ICP buffer size (the maximum actual data size)
DLI header size	96 bytes[2]	If your application uses the DLI, the buffer size must include this DLI header size

[1] For BSC, the protocol-specific DLI parameter is msgBlkSize (default is 1024 bytes).

[2] On most client platforms the DLI header is 76 bytes; however, this size is platform dependent. For initial installations Protogate recommends assuming a DLI header size of 96 bytes to calculate buffer sizes in the configuration files.

Step 3: Make the required changes to the protocol-specific portion of the client DLI configuration file as shown in Figure 2–1.

Step 4: Make the required changes to the client TSI configuration file as shown in Figure 2–2.

```
main                                // DLI “main” section:                //
{
        …
}
Session1                            // Session-specific parameters        //
{
        …

// BSC protocol-specific parameters for Session1:                         //

        msgBlkSize = 1500;
        …
}                                   // End of Session1 parameters         //
```

Figure 2–1: Client DLI Configuration File Changes (BSC Example)

```
main                                // TSI “main” section:                //
{
        maxBufSize = 1596  ;        // Must be 1596 (or greater)          //
        …
}
Conn1                               // Connection-specific parameters     //
{
        maxBufSize = 1596;
        …
}
```

Figure 2–2: Client TSI Configuration File Changes

Step 5: Make the required changes to the server MuxCfg TSI configuration file (located in your boot directory) as shown in Figure 2–3.

```
main                              // MuxCfg "main" section:            //
{
        maxBufSize = 1596  ;      // Must be 1596 (or greater)         //
        …
}
MuxConn1                          // Connection-specific parameters    //
{
        maxBufSize = 1596;
        …
}
```

Figure 2–3: Server MuxCfg TSI Configuration File Changes

2.4.2 Client TSI Buffer Configuration

For users who need to understand the details of the buffer management system, review Section 2.4.2 through Section 2.4.6 carefully. After you define the ICP buffer size as described in *Step 1* on page 41, the next step is to define the client TSI buffers.

The TSI provides its own buffer management scheme. Definitions in the client TSI configuration file allow you to create fixed-sized buffers in a TSI-controlled buffer pool (see Section 2.4.2.1). Each connection can then optionally be assigned a unique maximum buffer size (see Section 2.4.2.2). TSI applications can then access these buffers using the tBufAlloc and tBufFree TSI functions.

Note

For applications using Protogate's data link interface, the DLI uses the TSI buffer management system for its own buffer management. The dlBufAlloc and dlBufFree DLI functions provide access to buffers in the TSI buffer pool.

Your application is not required to use the TSI buffer management facilities, but Protogate highly recommends it for the following reasons:

- TSI allocates all buffers up front, resulting in better real-time performance than allocation through C malloc and free functions
- The number of TSI buffers is configurable for operating environments with limited system resources
- TSI allocates the buffer pool on boundaries which minimize memory access overhead
- TSI overhead is invisible to the user

2.4.2.1 TSI Buffer Pool Definition

The TSI buffer pool is configured through two parameter definitions in the "main" section of the client TSI configuration file. The maxBufSize parameter specifies the maximum size of each buffer in the TSI buffer pool. The maxBuffers parameter specifies the maximum number of buffers available in the TSI buffer pool and must support the maximum number of I/O requests that could be outstanding at any one time. After adjusting maxBufSize as described below, the product of the maxBufSize and maxBuffers parameters defines the TSI buffer pool size.

The maxBufSize parameter defines the maximum size of each buffer. This is the actual data size the TSI user application has available for its own use. When the buffer pool is defined, TSI calculates an "effective" buffer size which is maxBufSize plus the additional bytes required for a TSI header plus any alignment bytes. Alignment bytes are required only if the value of maxBufSize plus the TSI header bytes is not divisible by 4.

This "effective" buffer size is invisible to the user application (regardless of whether it interfaces to the DLI or the TSI); all interactions with the TSI buffer management facilities are based on maxBufSize and the connection-specific parameter described in Section 2.4.2.2. If you define maxBufSize as 1000 bytes, TSI assures that the buffer pool can provide 1000 bytes for TSI application data.

Figure 2–4 illustrates an example buffer calculation assuming the following sizes:

- maxBufSize is 1000 bytes
- The TSI header is 18 bytes
- The necessary alignment to make the total divisible by 4 is 2 bytes

TSI adds 18 bytes to the maxBufSize value to include the TSI header, making the actual size of the buffer allocated by TSI 1018 bytes. Because this actual size is not divisible by 4, TSI increments the value to the next modulo-4 value, in this case, 1020. Regardless of the final size, your TSI application has control of only maxBufSize bytes.

The TSI application program can obtain the value of maxBufSize using a tPoll request for the system configuration. Refer to the TSI_POLL_GET_SYS_CFG option (described in the *Freeway Transport Subsystem Interface Reference Guide*), which returns the iMaxBufSize field.

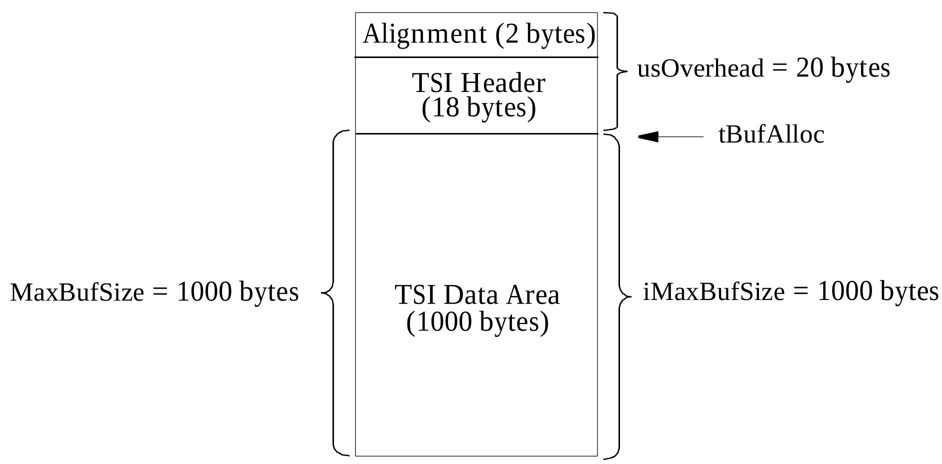

Figure 2–4: TSI Buffer Size Example

Note

The Figure 2–4 example, as viewed from the DLI application's perspective is shown in Figure 2–5. Of the 1000 bytes specified by the TSI maxBufSize parameter, 76 bytes are required for the DLI header. After calling dlOpen, the DLI application program can call dlPoll with the DLI_POLL_GET_SESS_STATUS option, which returns the usMaxSessBufSize field. This value is the actual data size available to the DLI application (924 bytes in the Figure 2–5 example).

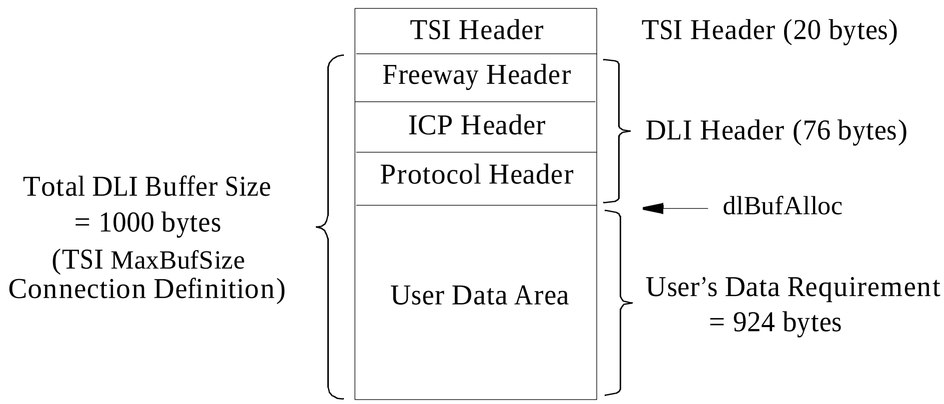

Figure 2–5: DLI Buffer Size Example

2.4.2.2 Connection-Specific Buffer Definition

After the TSI buffer pool is defined, you have the option of defining a unique maximum buffer size for each connection in the client TSI configuration file. If undefined, the connection buffer size defaults to the maxBufSize "main" definition for the TSI buffer pool described in the previous Section 2.4.2.1.

Note

The maximum connection buffer size should be at least as large as the defined ICP buffer size, plus any additional client requirements. For example, if you are using the DLI, you must include DLI overhead bytes in the total size of the application data area (see Figure 2–5).

To define a unique buffer size for a connection, use the *connection-specific* maxBufSize parameter. This connection buffer size is the buffer size the system allows the user for tWrite requests. No connection buffer size can be larger than maxBufSize defined for the TSI buffer pool.

The connection buffer size does not change the actual size of the buffer (actual buffers are all maxBufSize as defined for the TSI buffer pool); it only limits the acceptable size of application write buffers given to TSI through a tWrite request. It enforces a maximum data size that can be sent to the server in any one tWrite request. The tWrite function returns a TSI_WRIT_ERR_INVALID_LENGTH error if the write is attempted with a buffer exceeding the connection's maximum buffer size.

The tRead requests are not limited by the connection buffer size. The size of read requests, when using tRead, is defined by maxBufSize for the TSI buffer pool (in the "main" definition of the TSI configuration file).

2.4.2.3 TSI Buffer Size Negotiation

A connection's maximum buffer size can be changed "silently." When the client's connection to the Freeway server is accomplished, the client TSI and the server TSI negotiate a maximum buffer size for the established connection. If the sizes are different, the side with the larger connection buffer size changes its size to that of the smaller. After the connection is established, the negotiated maximum buffer size is available using a tPoll request for connection status. Refer to the TSI_POLL_GET_CONN_STATUS option (described in the *Freeway Transport Subsystem Interface Reference Guide*), which returns the usMaxConnBufSize field. Note that this "final" size is not available until the connection has been successfully established.

Note

The DLI application program can obtain the actual data size (after the TSI negotiation process during dlOpen) using a dlPoll request with the DLI_POLL_GET_SESS_STATUS option, which returns the usMaxSessBufSize field. See the example program in Section 5.5 on page 182.

2.4.3 Server TSI Buffer Configuration

After defining the ICP buffers and the client TSI buffers, the final step is to define the server TSI buffers. The same TSI buffer management design details apply to the server TSI buffers that were described in Section 2.4.2 on page 45 for the client TSI buffers. The only difference is that the server buffer definitions are specified in the MuxCfg server TSI configuration file, which is located in your boot directory. As with the client, define the TSI buffer pool size in the MuxCfg file's "main" section. Then define the optional connection-specific maximum buffer size for each connection. Simply define the connection buffer size for the largest associated client requirement. The buffer pool size must be at least as large as the largest connection buffer size. The *Freeway Transport Subsystem Interface Reference Guide* discusses the MuxCfg file in detail. Refer back to Section 2.4.1.1 on page 42 for a sample calculation of ICP, client, and server buffer sizes.

2.4.4 Buffer Allocation and Release

The TSI application obtains a buffer from the TSI buffer pool using the tBufAlloc function. The returned buffer address points to the available data area as shown in Figure 2–4 on page 47. The size returned is always the maxBufSize defined for the buffer pool. While the entire data area is available for user data, note the restrictions discussed previously in Section 2.4.2.2 regarding limits placed on tWrite requests by the connection's maximum buffer size definition. The user application releases a buffer back to the TSI buffer pool using the tBufFree function.

Note

DLI applications use the dlBufAlloc and dlBufFree functions to access buffers in the TSI buffer pool.

2.4.5 Cautions for Changing Buffer Sizes

If you need to change the buffer size of your application, keep the following cautions in mind:

- If you increase the ICP buffer size, there may be corresponding changes required in the client and server buffer sizes.
- If you have limited resources and increase the client or server maxBufSize parameter, consider decreasing the number of buffers allocated in the buffer pool (the maxBuffers parameter in the client TSI configuration file and the server MuxCfg file).
- Client read buffers too small for an inbound data buffer are returned to the client application with a TSI_READ_ERR_OVERFLOW error indication. Write requests with buffers too large are returned with a TSI_WRIT_ERR_INVALID_LENGTH error indication.

2.4.6 Using Your Own Buffers

If your DLI application needs to use its own buffers, it must know the exact number of overhead bytes used to store the TSI and DLI header information. Your application should call dlPoll to get the DLI system configuration information (Section 4.10 on page 114) so that it can allocate buffers correctly. Each buffer must be at least iMaxBufSize + usOverhead bytes in size (these values are described on page 79). Your application must give DLI the address of the memory buffer that is at usOverhead bytes from the beginning of the data area. Figure 2–6 shows a comparison of using the “C” malloc function versus the DLI dlBufAlloc function for buffer allocation. Figure 2–7 is a “C” code fragment demonstrating the use of the malloc function.

Note

For information about using your own buffers in a TSI application, see the *Freeway Transport Subsystem Interface Reference Guide*.

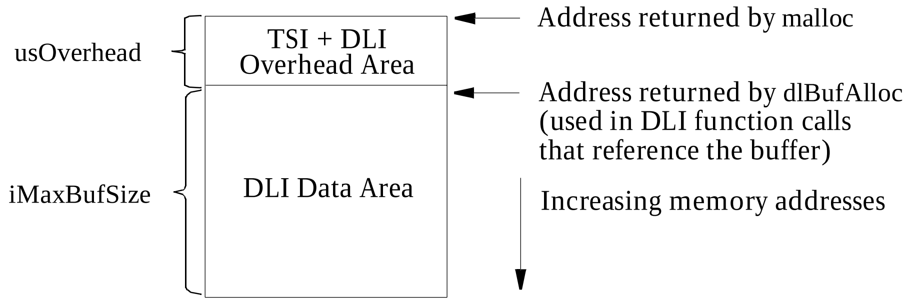

Figure 2–6: Comparison of malloc and dlBufAlloc Buffers

```
...
PCHAR              pBuf;
DLI_SYS_CFG        sysCfg;
int                iBufSize, iSessID;
...
dlPoll (0, DLI_POLL_GET_SYS_CFG, (PCHAR*)NULL, (PINT)NULL,
              (PCHAR)&sysCfg, (PDLI_OPT_ARGS*)NULL);
iBufSize = (int) sysCfg. usOverhead + sysCfg. iMaxBufSize;
pBuf = (PCHAR) malloc (iBufSize);
...
dlWrite (iSessID, &pBuf[sysCfg. usOverhead], 100,
              DLI_WRITE_NORMAL, (PDLI_OPT_ARGS*)NULL);
...
```

Figure 2–7: Using the malloc Function for Buffer Allocation

2.5 System Resource Requirements

2.5.1 Memory Requirements

Since the DLI operates on the TSI service layer, you must consider TSI resource requirements as well as DLI system resource requirements. For more information on calculating the TSI system resource requirements, refer to the *Freeway Transport Subsystem Interface Reference Guide*. The DLI system requirements can be calculated as follows:

Total memory requirements = program size
+ (number of buffers x size of buffer)
+ (number of sessions x 300)
+ (number of sessions x size of I/O queues x 44)
+ 32,000

Where:

- "number of buffers" is defined by the TSI maxBuffers parameter (page 148)
- "size of buffer" is defined by the TSI maxBufSize parameter (page 148)
- "number of sessions" is defined by the DLI maxSess parameter (page 63)
- "size of I/O queues" is defined by the sum of the DLI maxInQ parameter (page 64) and the DLI maxOutQ parameter (page 65)

2.5.2 Signal Processing

Both the DLI and TSI disable all signals during processing. The signals are ultimately delivered when they are re-enabled at the end of the DLI or TSI call. If this constraint causes a problem for your client application, consider implementing one of the following:

- use non-blocking I/O as described in Section 2.2 on page 32
- use the timeout TSI configuration parameter (page 149)

Under VMS, ASTs are disabled instead of signals.

Chapter 3 DLI Configuration

Note

The DLI configuration in this chapter also applies to an embedded ICP using the DLITE interface. If you are using an embedded ICP, also refer to the user guide for your ICP and operating system for configuration specific to DLITE.

3.1 Configuration Process Overview

The data link interface (DLI) consists of two major components:

- The dlicfg configuration preprocessor program defines the DLI environment prior to run time, using a text configuration file that you create or modify.
- The DLI reference library is used to build your DLI application. The DLI uses the transport subsystem interface (TSI).

The advantage of using the dlicfg preprocessor program is that you do not have to rebuild your application when you redefine the DLI or TSI environment.

The DLI and TSI configuration process is a part of the installation procedure and the loopback testing procedure described in the *Freeway User Guide*. However, during your client application development and testing, you might need to perform DLI and TSI configuration repeatedly.

The DLI and TSI configuration process is summarized as follows:

1. Create or modify a text file specifying the DLI session configuration for all ICPs and serial communication links in your Freeway system. Refer to your particular protocol programmer's guide for the protocol-specific link configuration options (if you need to change the default values).

2. Create or modify a text file specifying the configuration of the transport subsystem interface (TSI) connections.

3. Execute the dlicfg preprocessor program with the text file from Step 1 as input. This creates the DLI binary configuration file. If the optional DLI binary configuration filename is supplied, the binary file is given that name plus the .bin extension. If the optional filename is not supplied, the binary file is given the same name as your DLI text configuration file plus the .bin extension.

   dlicfg *DLI-text-configuration-filename [DLI-binary-configuration-filename]*

4. Execute the tsicfg preprocessor program with the text file from Step 2 as input. This creates the TSI binary configuration file. If the optional TSI binary configuration filename is supplied, the binary file is given that name plus the .bin extension. If the optional filename is not supplied, the binary file is given the same name as your TSI text configuration file plus the .bin extension.

   tsicfg *TSI-text-configuration-filename [TSI-binary-configuration-filename]*

Note

You must rerun dlicfg or tsicfg whenever you modify the text configuration file so that the DLI or TSI functions can apply the changes.

When your application calls the dlInit function, the DLI and TSI binary configuration files are used to configure the DLI sessions and TSI connections.

Note

The *Freeway User Guide* describes the make files and command files provided to automate the above process and copy the resulting binary configuration files to the appropriate directories. Additionally, each protocol programmer's guide describes the related protocol specifics of the DLI/TSI configuration process.

3.2 DLI Configuration versus TSI Configuration

As shown in Step 2 and Step 4 of the previous Section 3.1, the transport subsystem interface (TSI) configuration is an integral part of the overall DLI configuration process (also see Figure 3–1).

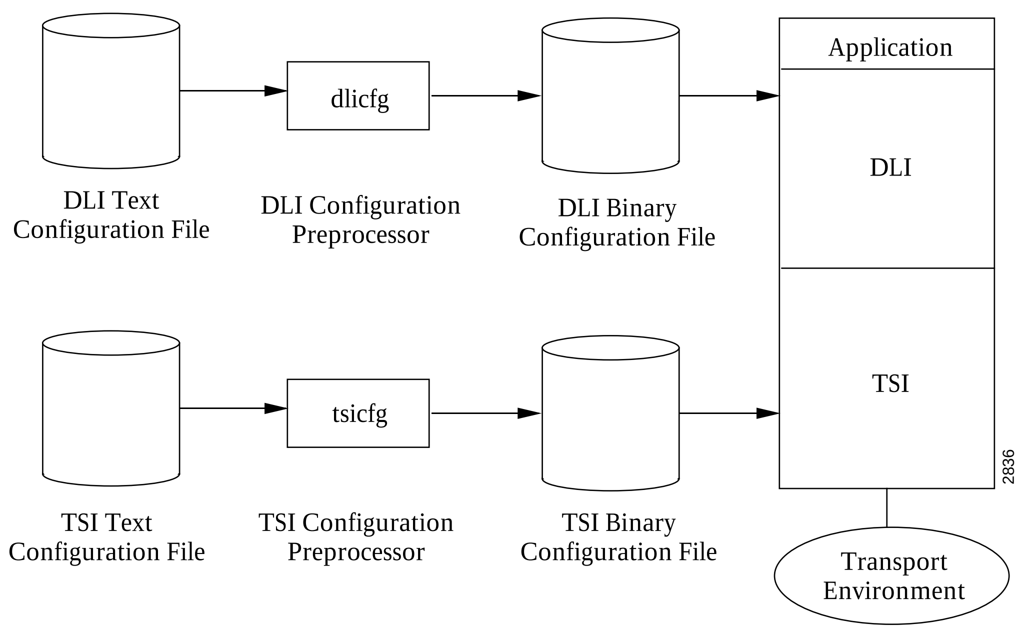

Figure 3–1: DLI Overall Architecture

The main distinction between the DLI and TSI configuration processes is that the DLI configures *sessions* associated with ICP links, and the TSI configures transport *connections* for the DLI sessions. Otherwise, the TSI configuration process is very similar to the DLI configuration process. Both processes use a preprocessor program (dlicfg or tsicfg) to translate your text configuration file into a binary configuration file. In both cases, the text file consists of a "main" section to configure parameters independent of sessions or connections, plus additional sections to configure individual sessions or connections.

The tutorial example program in Chapter 5 gives example DLI and TSI text configuration files to support the example program. Refer to the *Freeway Transport Subsystem Interface Reference Guide* for the TSI configuration details beyond the scope of the example program. If your application uses the DLI functions, you do not need an in-depth understanding of the TSI functions. However, if your application requires that you access the TSI functions directly, you need a detailed understanding of the *Freeway Transport Subsystem Interface Reference Guide*.

3.3 Introduction to DLI Configuration

The dlicfg program is a configuration preprocessor that translates a DLI text configuration file into a binary configuration file. During the translation process, dlicfg verifies and processes each configuration entry in the text configuration file, and the results are stored in the binary configuration file. This process ensures the validity of the configuration parameters before their use by the DLI reference library. The DLI configuration services provide the following features:

- Free-formatted configuration language
- Informative parameter names
- Procedure-like definition entry for each session definition
- Extensive syntax checking capability

- Extensive semantic checking capability
- Session-based definition capability
- Use of CCITT CRC-16 to detect any corruption of the binary configuration file

The DLI reference library is a set of function calls used by applications to exchange data between two or more locations in a Freeway-supported network on a well-defined data link protocol (i.e. BSC, FMP, X.25, etc.). The DLI reference library uses the DLI binary configuration file to configure DLI services as well as sessions managed by the DLI. Together with the DLI configuration services, the DLI reference library provides data link applications with a flexible network programming environment.

3.3.1 DLI Configuration Language

Each session definition entry in the DLI text configuration file defines a unique data link session to be established between your DLI application and a remote data link application. Refer to Section 3.5.2 for details of the language grammar. The DLI configuration can be described as follows:

```
session-name
{
        parameter-name = parameter-value;          // comments are ignored      //
}
```

Each session-name you choose must uniquely identify a session within the same configuration file; dlicfg makes no attempt to ensure the uniqueness of names within the same configuration file. Each parameter-name is uniquely defined by dlicfg. Comments are considered white spaces and are ignored by dlicfg.

3.3.2 Rules of the DLI Configuration File

A session name or a parameter name must adhere to the following naming rules:

1. It is similar to variable names in the C language.

2. It can be a string of alphabetic (A through Z, a through z, and _) and numeric (0 through 9) characters.

3. The first character must be alphabetic.

4. The length must not be more than 20 characters.

5. A session name is case-sensitive while a parameter name is not.

6. The first definition entry can be defined as “main” for the DLI main configuration parameters. If no “main” entry is defined as the first definition entry, a default “main” entry is defined and included as the first entry in the binary configuration file.

7. A session name must be unique within the same configuration file. Otherwise, DLI selects the first one of the identical session names.

8. DLI does not verify the duplication of session definition entries at the session level or at the parameter level. That means if you have defined the same session entry more than once, the first one is used. If you have defined a parameter within a session definition entry more than once, the last value is used.

3.3.3 Binary Configuration File Management

The binary configuration file is created in the same directory as the location of the text configuration file (unless a different path is supplied with the optional filename described in Section 3.1 on page 55). On all but VMS systems, if a file already exists in that directory with the same name, the existing file is renamed by appending the .BAK extension. If the renamed file duplicates an existing file in the directory, that existing file is removed by the configuration preprocessor program.

Note

The default binary configuration name contains the period ‘.’ character which plays a special role in the processing of the configuration files. See Section 3.3.4.

3.3.4 On-line Configuration File Processing

The DLI and TSI can perform the configuration processing on-line. While this feature is available, Protogate recommends adherence to the off-line configuration file process previously described in Section 3.1 on page 55, which is better managed and slightly more efficient.

The off-line process can be performed on-line during DLI and TSI initialization (dlInit) by providing a configuration filename without an embedded '.' character. When such a filename is recognized, the DLI/TSI attempts to open the file as a text file and calls the DLI configuration preprocessor program (dlicfg). The output file is named "filename".bin. An error in the configuration file aborts the dlInit processing with an appropriate error in the DLI/TSI log file.

This on-line method requires the configuration text files and the dlicfg and tsicfg pre-processor programs to reside in the same directory as the application executable. The resulting .bin file is placed in this same directory.

Note

Unless on-line configuration is desired, be sure a '.' character appears in the configuration filename provided to dlInit.

3.4 DLI Session Definition

The information exchange between your application and the DLI is managed by a *session*. A session allows your application to communicate with one serial link on one ICP. A separate session is required for each serial link on each ICP, though for some protocols multiple sessions can be defined for a link. Associated with each session are client parameters such as queue lengths plus the protocol-specific link configuration parameters (described in your particular protocol programmer's guide).

Two types of configuration sections are included in the DLI text configuration file. The first section (called “main”) specifies the configuration for non-session-specific operations. Subsequent sections define one or more specific sessions.

The dlicfg program processes your DLI text configuration file and creates a DLI binary configuration file. Your application then specifies the binary configuration filename as a parameter when it calls the DLI initialization function, dlInit.

3.4.1 DLI “main” Configuration Section

The first section in the DLI text configuration file, which is called “main,” specifies the DLI configuration for non-session-specific operations. Figure 3–2 is an example of the “main” section. Notice that the DLI “main” section must specify the TSI binary configuration filename (the tsiCfgName DLI parameter) if it is different from the default name. If you use all the default values, the “main” section is optional.The “main” DLI parameters are shown in Table 3–1, along with the defaults. You need to include only those parameters whose values differ from the defaults.

```
main
{
    tsiCfgName = “tsisynccfg.bin”;          // TSI binary configuration file              //
}
```

Figure 3–2: DLI Example “main” Configuration Section

3.4.2 DLI Session Configuration Sections

Each additional section of the DLI text configuration file specifies a session associated with a particular ICP link (port). Each Freeway serial communication link can be configured independently of the other links. The parameters are divided into two groups: client-related parameters and protocol-specific link characteristics. Each DLI session has an associated TSI connection (the transport DLI parameter). The DLI client-related parameters are shown in Table 3–2, along with the defaults.

Table 3–1: DLI “main” Parameters and Defaults

Parameter	Default	Valid Values	Description
asyncIO	“no”	boolean	A value of “no” specifies blocking I/O. If set to “yes”, the TSI must also be configured for non-blocking I/O in order for this flag to be effective
callbackQsize	500	1–5000	The size of an internal DLI queue that saves callback requests. If this queue overflows, a DLI_CALLBACK_Q_OVRFLOW error (page 188) is saved in the DLI log file, and the application’s callback might be lost. Change this value with caution. Users with heavy I/O requirements should examine the DLI log file during development for evidence of this error.
logLev	0	0–7	An integer value defining the level of logging DLI performs. 0 = no logging; 1 = most severe; 7 = least severe
logName	“dlilog”	string (ð 255)	A string of characters defining the name (path) of the file to store the DLI logging information. To direct logging information to the screen, define logName to be stdout. If the path is not included, the current directory is assumed.
maxSess	128	1–1024	An integer value that defines the maximum number of sessions DLI can manage at the same time
sessPerConn	16	1–16	An integer value specifying the number of sessions that are allowed to operate simultaneously on one transport connection
traceLev	0	0–31	An integer value defining the level of tracing (or the sum of several levels) which the DLI performs for this session. See also Appendix D. 0 = no trace 1 = read only 2 = write only 4 = interrupt only 8 = application IOCH 16 = user’s data
traceName	“dlitrace”	string (ð 255)	A string of characters defining the name (path) of the file to store the DLI tracing information. If the path is not included, the current directory is assumed.
traceSize	0	512–1048576	An integer value defining the size of the trace file defined by the traceName parameter.
tsiCfgName	“tsicfg.bin”	string (ð 255)	A string of characters specifying the name (path) of the TSI binary configuration file. If the path is not included, the current directory is assumed. If the default names were not used in generating the binary files, ensure correct use of the ‘.’ character (Section 3.3.4 on page 61).

Table 3–2: DLI Client-Related Parameters and Defaults

Parameter	Default	Valid Values	Description
alwaysQIO	“no”	boolean	Specifies whether or not the DLI always queues the I/O request. If “yes” then the request is queued even if it can be satisfied immediately. Setting alwaysQIO to “yes” could ease your application implementation.
asyncIO	“no”	boolean	A value of “no” specifies blocking I/O. If set to “yes”, the TSI must also be configured for non-blocking I/O in order for this flag to be effective
boardNo	0	0–128	An integer value specifying a particular ICP within the locally attached Freeway to be used for this session
cfgLink	“yes”	boolean	Specifies whether or not DLI configures the link before opening it
enable	“yes”	boolean	If set to “yes,” dlOpen also enables the ICP link.
family	“protocol”	string (ð 20)	A string of characters specifying the protocol family to be used
localAck	“yes”	boolean	Specifies whether or not DLI should handle the protocol-specific local data acknowledgment. The current DLI implementation uses the user’s buffer to receive the localAck packet to be processed by DLI. If your application needs to see this local acknowledgment message, set the localAck parameter to “no.” Your application must then read the localAck message using the dlRead function with the optional arguments parameter (*Raw* operation). *Note:* When using non-blocking I/O, there must be at least one outstanding read request before DLI receives the localAck packet from the ICP.
logLev	0	0–7	An integer value specifying the level of logging DLI performs for this session. If specified, this value overrides the logLev parameter in the “main” section. 0 = no logging; 1 = most severe; 7 = least severe.
maxErrors	100	10–100	An integer value specifying the maximum number of consecutive I/O errors DLI can tolerate before declaring the session is unusable
maxInQ	10	2–100	An integer value specifying the maximum number of entries allowed in the DLI internal input queue

Table 3–2: DLI Client-Related Parameters and Defaults *(Cont'd)*

Parameter	Default	Valid Values	Description
maxOutQ	10	2–100	An integer value specifying the maximum number of entries allowed in the DLI internal output queue
mode	"shrmgr"	string (ð 20)	A string of characters specifying the protocol-specific access mode when DLI interacts with the ICP. Refer to your particular protocol programmer's guide.
portNo	0	0–64	An integer value specifying a particular link in the above defined ICP (boardNo) to be used for this session
protocol	no default	"raw" or a specific protocol ("BSC3780", "FMP", etc.)	A string of characters (maximum of 20) specifying "raw" (*Raw* operation) or the data link protocol (*Normal* operation) for this session. This is a required parameter.
reuseTrans	"no"	boolean	Specifies whether or not DLI reuses the existing transport connection for the same session services. This is to declare the use of multiple sessions per connection capability provided by DLI
traceLev	0	0–31	An integer value defining the level of tracing (or the sum of several levels) which the DLI performs for this session. If specified, this value overrides the "main" traceLev parameter. See also Appendix D. 0 = no trace 1 = read only 2 = write only 4 = interrupt only 8 = application IOCH 16 = user's data
transport	no default	string (ð 20)	A string of characters specifying the connection name defined in the TSI configuration file to be used by this session. This is a required parameter.

The example shown in Figure 3–3 on page 67 defines one session for *Raw* operation and one session for *Normal* operation using the FMP protocol. The parameter names are not case sensitive; the definition in upper and lower case is for readability only. You need to include only those parameters whose values differ from the defaults. The protocol-specific parameters, which would follow at the end of the session definition as indicated at the end of the figure, are described in your particular protocol programmer's guide.

3.4.3 Protocol-Specific Parameters for a Session

See your particular protocol programmer's guide for information on the protocol-specific ICP link configuration parameters. The parameters listed in Table 3–3 are included here since they are used by most protocols; however, your particular protocol software might have protocol-specific configuration methods for these parameters.

Table 3–3: DLI Protocol-Specific ICP Link Configuration Parameters

Parameter	Default	Valid Values	Most Common Usage
msgBlkSize	1024	256–8192	Allows DLI to configure ICP message buffer size (applies to all links on an ICP)
writeType	“Normal”	string (ð20) (protocol specific)	A string of characters specifying the type of data to be exchanged when using *Normal* operation (Section 2.3 on page 34)

```
//---------------------------------------------------------------------//
// This line is a comment line, ignored by dlicfg                                    //
//---------------------------------------------------------------------//

main                                          // DLI “main” section:          //
{
        tsiCfgName = “tsiCfg.bin”;            // TSI binary config file       //
        maxSess = 256;
        asyncIO = “Yes”;                      // Non-blocking I/O             //
        logLev = 3;
        traceLev = 4;
        traceSize = 64000;
        traceName = “stdout”;
        logName = “stdout”;
}
//-----------------------------------------------------------------//
// This configuration section defines a generic session for a       //
// raw operation interface to the Freeway system.  The data link   //
// protocol-specific parameters are not defined in this example.    //
//-----------------------------------------------------------------//

RawSess1      // First session name: raw operation on link 1 of ICP 0          //
{
        protocol = "Raw";                     // Raw session type             //
        transport = "FW1";                    // Transport connection name    //
                                              // defined in TSICfgName file   //
        mode = "User";                        // Access mode for ICP          //
        family = "Protocol";                  // Family -- Protocol only      //
        boardNo = 0;                          // ICP board number -- based 0  //
        portNo = 1;                           // Link number                  //
        traceLev = 1;
        logLev = 0;
        maxInQ = 20;                          // Max # entries in input Q     //
        maxOutQ = 20;                         // Max # entries in output Q    //
        maxErrors = 100;
        localAck = "Yes";
        asyncIO = "yes";                      // Non-blocking I/O             //
}
```

Figure 3–3: DLI Configuration Text File for Two Links

```
link0icp1        // Second session name: FMP session at link 0 on ICP 1           //
{
        protocol = "FMP";
        transport = "tcp2";
        mode = "user";              // Access mode for ICP                         //
        family = "protocol";
        boardNo = 1;                // ICP 1.                                      //
        portNo = 0;                 // link 0 on that ICP.                         //
        logLev = 0;                 // NO log....                                  //
        traceLev = 0;               // NO trace either...                          //
        maxInQ = 20;
        maxOutQ = 30;
        maxErrors = 100;            // reject request after 100 errors!            //
        localAck = "Yes";           // implement Local ack                         //
        cfgLink = "Yes";            // configure link prior to its use.            //
// Optional protocol-specific parameters start here:                               //
                .
                .
                .
}
```

Figure 3–3: DLI Configuration Text File for Two Links *(Cont'd)*

3.5 Miscellaneous DLI Configuration Details

After you are familiar with the fundamentals of working with the dlicfg preprocessor program, the additional details described in this section might be of interest.

3.5.1 DLI Configuration Error Messages

The dlicfg program can display one of the error or warning messages listed below. Refer to Table 3–1 on page 63 and Table 3–2 on page 64 for the DLI configuration parameter descriptions.

Invalid type specified — STRING expected   Your parameter value does not match the expected type. *Action:* Review your text configuration file for errors, and try again.

Invalid type specified — BOOLEAN expected   You must use a Boolean value (“yes” or “no”) for this parameter. *Action:* Review your text configuration file for errors and try again.

Invalid type specified — DEC/HEX/OCT expected   The expected type is decimal, hexadecimal, or octal data format. *Action:* Review your text configuration file for errors and try again.

Invalid type specified — FLOAT expected   The expected type is floating point format. *Action:* Review your text configuration file for errors and try again.

Invalid range specified   The provided parameter value is out of range. *Action:* Review your text configuration file for errors and try again.

Internal error!   This is an internal error in the dlicfg program. *Action:* Rerun dlicfg with your text configuration file. If this error consistently occurs, save your text configuration file and contact Protogate for further assistance.

No “main” — Default is used   This is a warning message that your text configuration file does not have the “main” section specified as the first entry. *Action:* None if you do not wish to define the “main” section yourself. Otherwise, consider adding the “main” section as the very first section in the DLI text configuration file.

Redefined “main” — Definition ignored This is a warning message. Either you defined the “main” section twice or you did not code the “main” section as the very first entry in your DLI text configuration file. *Action:* Review your text configuration file, correct the problem, and rerun dlicfg.

Invalid session name You specified a protocol parameter value (page 65) that is not recognized by DLI. *Action:* Review your text configuration file. Correct the error and try again.

Undefined parameter name The provided parameter name is not defined. *Action:* Review your text configuration file for errors and try again.

Invalid parameter for specified protocol This parameter does not belong to this protocol. *Action:* Refer to your particular protocol programmer’s guide, review your text configuration file for errors, and try again.

Invalid mode specified The ICP mode parameter (page 65) is invalid. *Action:* Refer to your particular protocol programmer’s guide for the valid access modes, review your text configuration file for errors, and try again.

Invalid protocol family specified The family parameter value (page 64) is undefined. *Action:* Review your text configuration file for errors and try again.

Failed processing file dlicfg failed to complete processing your configuration file. *Action:* Review your text configuration file for errors and try again.

syntax error - cannot backup This is an internal LEX/YACC error. *Action:* Retry the operation.

out of memory This is an internal LEX / YACC error. *Action:* Retry the operation.

yacc stack overflow   This is an internal YACC error. *Action:* Retry the operation.

syntax error   A syntax error was encountered in your text configuration file. *Action:* Locate and correct the error and try the operation again.

3.5.2 Protogate Definition Language (PDL) Grammar

The following *extended BNF* metalanguage describes the language used to create the DLI text configuration file. The following is a brief description of the symbols used:

1. A string inside of <> is a *non-terminal* symbol. Its definition is located somewhere down the list.
2. Strings inside of {} separated by a vertical bar (|) make up a list of options. You can select one or none of the options.
3. A string inside of [] is an optional string.
4. *Terminal* symbols are those not surrounded by <>.

Context Free Grammar

<config_entry> ::= <session_name> <leftbr> <config_stmt_list> <rightbr>

<session_name> ::= <identifier>

<config_stmt_list> ::= <config_stmt>{<config_stmt_list>}

<config_stmt> ::= [<parameter_name> <equal><parameter_value>;]

<paramter_name> ::= <identifer>

<parameter_value> ::= {<string> | 0x<hex> | <decimal> | 0<octal>
0b<binary> | <float>}

<string> ::= <doublequote><str><doublequote>

<str> ::= [<char>{<str>}]

<decimal> ::= <decdigit>[<decimal>]

<octal> ::= <octdigit>[<octal>]

<binary> ::= <bindigit>[<binary>]

<hex> ::= <hexdigit>[<hex>]

<float> ::= <decimal>.<decimal>

<equal> ::= =

<leftbr> ::= {

<rightbr> ::= }

<doublequote> ::= "

<char> ::= 1..255

<decdigit> ::= 0..9

<hexdigit> ::= <decdigit>, a–f

<octdigit> ::= 0..7

<bindigit> ::= 0..1

<alpha> ::= a–z, A–Z, _

<digit> :: <decdigit>

<identifer> ::= <alpha>[<restid>]

<restid> ::= <alphadigit>[<restid>]

<alphadigit> ::= <alpha> | <digit>

Chapter 4

DLI Functions

Note

The DLI functions in this chapter also apply to an embedded ICP using the DLITE interface. If you are using an embedded ICP, also refer to the user guide for your ICP and operating system for functions specific to DLITE.

4.1 Overview of DLI Functions

This chapter describes the data link interface (DLI) functions used by your application to interface to Freeway's supported data link protocols. After you are familiar with the function calls, Chapter 5 presents some tutorial example programs to help you write your application.

The DLI shields your application from the detailed interfaces between your Freeway server and your operating environment. These detailed interfaces include network transport protocol, data exchange protocol between your application and the Freeway server, and the intelligent communication processors (ICPs).

The DLI is provided as a C library to link with your application. Appendix A describes the header files that your application needs to include at compilation time.

4.1.1 DLI Error Handling

The dlerrno variable is globally available to your application and offers similar services to errno provided in the C language. DLI uses dlerrno to store all its error codes. Your application should check this value on all returns from DLI function calls (see the dlpErrString function described in Section 4.9 on page 113). Applicable error codes are listed with each function call described in this chapter. Appendix B gives a complete list of DLI error codes.

Note

While developing your DLI application, if a particular error occurs consistently, contact Protogate for further assistance.

4.1.2 Overview of DLI Functions

After the protocol software is downloaded to the Freeway ICP, the client and Freeway can communicate by exchanging messages. These messages configure and activate each ICP link and transfer data. The client application issues reads and writes to transfer messages to and from the ICP.

4.1.2.1 Categories of DLI Functions

The DLI library functions are categorized as shown in Table 4–1.

Table 4–1: DLI Function Categories

Category	DLI Functions	Usage
Preparation and termination	dlInit, dlTerm	Initialize and terminate DLI services
Session handling	dlOpen, dlListen, dlClose	Establish and terminate a session with a remote data link application
Data transfer	dlRead, dlWrite, dlPoll, dlPost[a], dlpErrString, dlSyncSelect	Exchange data with a remote application and obtain status or error information related to the session
Control functions	dlControl	Reset/download ICP
Buffer management	dlBufAlloc, dlBufFree	Obtain and release fixed-size DLI buffers

[a] Server-resident application only

4.1.2.2 Summary of DLI Functions

The DLI functions used in writing a client application are presented alphabetically in Section 4.2 through Section 4.15. For easy reference after you are familiar with the details of each function call, Table 4–2 summarizes the DLI function syntax and parameters, listed in the most likely calling order.

Caution

When using non-blocking I/O, there must always be at least one dlRead request queued to avoid loss of data or responses from the ICP.

An overview of using the DLI functions is:

- Start up communications (dlInit, dlOpen, dlBufAlloc)
- Send requests and data using dlWrite
- Receive responses using dlRead
- For blocking I/O, use dlSyncSelect to query read availability status for multiple sessions
- For non-blocking I/O, handle I/O completions at the software interrupt level in the completion handler established by the dlInit or dlOpen function, or by periodic use of dlPoll to query the I/O completion status
- Monitor errors using dlpErrString
- If necessary, reset and download the protocol software to the ICP using dlControl
- Shut down communications (dlBufFree, dlClose, dlTerm)

Table 4–2: DLI Functions: Syntax and Parameters (Listed in Typical Call Order)

DLI Function	Parameter(s)	Parameter Usage
int dlInit (see page 99)	(char *cfgFile, char *pUsrCb, int (*fUsrIOCH)(char *pUsrCb));	DLI binary configuration file name Optional I/O complete control block Optional IOCH and parameter
int dlPost (see page 121)	(void);	
int dlListen (see page 103)	(char *cSessionName, int (*fUsrIOCH) (char *pUsrCB, int iSessionID));	Session name in DLI config file Optional I/O completion handler Parameters for IOCH
int dlOpen (see page 106)	(char *cSessionName, int (*fUsrIOCH) (char *pUsrCB, int iSessionID));	Session name in DLI config file Optional I/O completion handler Parameters for IOCH
int dlPoll (see page 114)	(int iSessionID, int iPollType, char **ppBuf, int *piBufLen, char *pStat, DLI_OPT_ARGS **ppOptArgs);	Session ID from dlOpen Request type Poll-type dependent parameter Size of I/O buffer (bytes) Status or configuration buffer Optional arguments for dlRead
int dlpErrString (see page 113)	(int dlErrNo);	DLI error number (global variable dlerrno)
char *dlBufAlloc (see page 86)	(int iBufLen);	Minimum buffer size
int dlRead (see page 122)	(int iSessionID, char **ppBuf, int iBufLen, DLI_OPT_ARGS *pOptArgs);	Session ID from dlOpen Buffer to receive data Maximum bytes to be returned Optional arguments structure
int dlWrite (see page 134)	(int iSessionID, char *pBuf, int iBufLen, int iWritePriority, DLI_OPT_ARGS *pOptArgs);	Session ID from dlOpen Source buffer for transfer Number of bytes to write Normal or expedite write Optional arguments structure

Table 4–2: DLI Functions: Syntax and Parameters (Listed in Typical Call Order) *(Cont'd)*

DLI Function	Parameter(s)	Parameter Usage
int dlSyncSelect (see page 128)	(int iNbrSessID, int sessIDArray[], int readStatArray[]);	Number of session IDs Packed array of session IDs Array containing read status for IDs
char *dlBufFree (see page 89)	(char *pBuf);	Buffer to return to pool
int dlClose (see page 95)	(int iSessionID, int iCloseMode);	Session ID from dlOpen Mode (normal or force)
int dlTerm (see page 132)	(void);	
int dlControl	(char *cSessionName, int iCommand, int (*fUsrIOCH) (char *pUsrCB, int iSessionID));	Session name in DLI config file Command (*e.g.* reset/download) Optional I/O completion handler Parameters for IOCH

4.1.3 DLI Data Structures

This section describes the following DLI data structures that your application can use:

- the DLI system configuration structure used by dlPoll
- the DLI session status structure used by dlPoll
- the DLI optional arguments structure used by dlRead and dlWrite for *Raw* operation

4.1.3.1 DLI System Configuration

After initializing the DLI services using dlInit, your application obtains system configuration parameters from DLI by calling dlPoll with the DLI_POLL_GET_SYS_CFG option (Section 4.10). The iMaxBufSize field reports the size of the buffers allocated in the TSI buffer pool (data size), and the usOverhead field reports overhead DLI requires in each data buffer (this size is actually the total overhead required, including both DLI and TSI requirements). This information is useful if your application uses its own buffers instead of DLI buffer management's (see Section 2.4.6 on page 51). Your application receives the system configuration information in the data structure shown in Figure 4–1. Table 4–3 describes the fields.

```
typedef struct          _DLI_SYS_CFG
{
   unsigned short       usMaxSess;          /* Max # of sessions defined      */
   unsigned short       usMaxBufs;          /* Max # of buffers defined       */
   unsigned short       usNumActiveSess;    /* # of cur. active sessions      */
   unsigned short       usNumBufsUsed;      /* # of buffers used              */
   unsigned short       usNumBufsAvail;     /* # of buffers avail             */
   unsigned short       usOverhead;         /* # of bytes for int bufs        */
   BOOLEAN              tfAsyncIO;          /* yes = non-blocking I/O         */
   int                  iMaxBufSize;        /* Max buffer size defined        */
      unsigned char     cTraceFileName[DLI_MAX_FILENAME+1]; /*trace file*/
}     DLI_SYS_CFG;
```

Figure 4–1: DLI System Configuration Data Structure

Table 4–3: DLI System Configuration Data Structure Fields

Field	Description
usMaxSess	The maximum number of sessions that can be active simultaneously. This value is configurable through the DLI configuration file. The parameter's name is maxSess in the "main" configuration section (page 63).
usMaxBufs	The maximum number of buffers available for your application. This value is configurable through the TSI configuration file. The parameter's name is maxBuffers in the "main" configuration section (page 148).
usNumActiveSess	The number of sessions currently in use. This number should be less than or equal to the usMaxSess value.
usNumBufsUsed	The number of buffers currently in use. This number should be less than or equal to the usMaxBufs above.
usNumBufsAvail	The number of buffers currently available for use. This number should be less than or equal to the usMaxBufs above.
usOverhead	The number of additional bytes that must precede your data area in a buffer that your application requests DLI to read or to write. Your application needs to be aware of this value only if it does not wish to use the DLI buffer management scheme (see Section 2.4 on page 40).
tfAsyncIO	A boolean value indicating the DLI was configured to use blocking or non-blocking I/O ("yes" = non-blocking I/O). If you require non-blocking I/O, both DLI and TSI must be configured for non-blocking I/O, else the default is blocking I/O.
iMaxBufSize	The maximum data length available in the TSI buffer pool.
cTraceFileName	The name of the file containing the trace after the application terminates normally.

4.1.3.2 DLI Session Status

To obtain information related to an active session, your application calls dlPoll with the DLI_POLL_GET_SESS_STATUS option (Section 4.10). This information contains the negotiated buffer size in the usMaxSessBufSize field, which is the actual data size available for this session's user data (Section 5.5 on page 182 gives an example program for using the usMaxSessBufSize field). You can get the session status information any time after a dlOpen returns successfully. Your application receives the session status information in the data structure shown in Figure 4–2. Table 4–4 describes the fields.

```
typedef struct          _DLI_SESS_STAT
{
    short               iQReadSize;             /* size of read/input q         */
    short               iQWriteSize;            /* size of write/output q       */
    short               iQNumRead;              /* # of entries in read q       */
    short               iQNumWrite;             /* # of entries in write q      */
    short               iQNumReadDone;          /* # of IO complete in read q */
    short               iQNumWriteDone;         /* # of IO complete in write q*/
    short               iMaxErrors;             /* max # of IO errs allowed     */
    short               iNumErrors;             /* # of IO errors               */
    short               iSessStatus;            /* current session status       */
    short               iICPMode;               /* ICP mode of operation        */
    short               iBoardNo;               /* ICP board number defined     */
    short               iPortNo;                /* ICP link number              */
    unsigned short      usMaxSessBufSize;       /* maximum user data area       */
    char      cServerVer[DLI_MAX_STRING+1];     /* Freeway Server version       */
}   DLI_SESS_STAT;
```

Figure 4–2: DLI Session Status Data Structure

Caution

Calling dlPoll with the DLI_POLL_GET_SESS_STATUS option is costly because it checks the entire input and output queues for I/O completion status; therefore, this call should be made sparingly.

Table 4–4: DLI Session Status Data Structure Fields

Field	Description
iQReadSize	The size of the input queue, configured using the maxInQ DLI configuration parameter (page 64) in the session definition section.
iQWriteSize	The size of the output queue, configured using the maxOutQ DLI configuration parameter (page 65) in the session definition section.
iQNumRead	The current number of read requests in the read queue. This value is less than or equal to iQReadSize.
iQNumWrite	The current number of write requests in the write queue. This value is less than or equal to iQWriteSize.
iQNumReadDone	The current number of read requests that are complete or timed out in the input queue. This value is less than or equal to iQReadSize. When using blocking I/O, this field is always zero and must not be used to determine when to queue a dlRead request.
iQNumWriteDone	The current number of write requests that are complete or timed out in the output queue. This value is less than or equal to iQWriteSize.
iMaxErrors	The maximum number of errors this session can tolerate before it rejects I/O requests from your application. This value is configured using the maxErrors DLI configuration parameter (page 64).
iNumErrors	The number of I/O errors for this session since the session establishment. Your application can monitor this value for the health of an active session.
iSessStatus	The current status of the session. The valid session status values are: DLI_STATUS_DEAD_SOCKET DLI has detected a failure on this session's connection to Freeway. Your application can retrieve any pending buffers and close the session (dlClose). Attempts to read or write to Freeway after this state is entered will result in failures (I/O requests are returned with the "…INVALID_STATE" error code). DLI_STATUS_FAILED The current DLI session is not available for use because it failed to establish a session with Freeway (*Raw* operation) or with the remote data link application (*Normal* operation). DLI_STATUS_NOT_READY DLI is still trying to establish a session with Freeway or with the remote data link application. DLI_STATUS_READY DLI successfully established a session with Freeway or the remote data link application. Your application can now read or write to this session.

Table 4–4: DLI Session Status Data Structure Fields

Field	Description
iICPMode	The mode parameter (page 65) of the ICP specified in the configuration for this session.
iBoardNo	The boardNo parameter (page 64) of the ICP specified in the configuration file.
iPortNo	The portNo parameter (page 65) of the ICP specified in the configuration file.
usMaxSessBufSize	The maximum buffer area available to the user for the transfer of data. This value originates with the buffer size defined in the connection definitions (TSI configuration file) associated with this session. It might be modified during the dlOpen process when the maximum buffer sizes are negotiated between the client TSI and the server TSI. This value includes the DLI overhead requirements.
cServerVer	A string containing the version of the Freeway server.

4.1.3.3 DLI Protocol-Specific Optional Arguments

If your data link application uses *Raw* operation (or a mixture of *Normal* and *Raw* operation), you must fully understand this important data structure, the data flow of the underlying data link protocol used by your application, and the internal architecture of the Freeway server (refer back to Figure 1–3 on page 26). The optional arguments structure is used by dlRead and dlWrite to pass the protocol-specific information required for *Raw* operation.

The DLI data format (which is internal to the DLI) includes a Freeway header, an ICP header, a Protocol header, and the data portion. The Freeway header is used exclusively between the DLI layer and the MsgMux component of Freeway. The ICP header and the Protocol header are used between the DLI layer and the ICP protocol service. The data portion is used between the DLI application and the remote data link application. The optional arguments data structure shown in Figure 4–3 below implements the Freeway DLI data format shown in Figure 4–4.

```
typedef struct            _DLI_OPT_ARGS
{
        unsigned short    usFWPacketType;     /* Server's packet type          */
        unsigned short    usFWCommand;        /* Server's cmd sent or rcvd     */
        unsigned short    usFWStatus;         /* Server's status of I/O ops*/
        unsigned short    usICPClientID;      /* old su_id                     */
        unsigned short    usICPServerID;      /* old sp_id                     */
        unsigned short    usICPCommand;       /* ICP's command.                */
        short             iICPStatus;         /* ICP's command status          */
        unsigned short    usICPParms[3];      /* ICP's extra parameters        */
        unsigned short    usProtCommand;      /* protocol command              */
        short             iProtModifier;      /* protocol cmd's modifier       */
        unsigned short    usProtLinkID;       /* protocol link ID              */
        unsigned short    usProtCircuitID;    /* protocol circuit ID           */
        unsigned short    usProtSessionID;    /* protocol session ID           */
        unsigned short    usProtSequence;     /* protocol sequence             */
        unsigned short    usProtXParms[2];    /* protocol extra parms          */
}       DLI_OPT_ARGS;
typedef DLI_OPT_ARGS *PDLI_OPT_ARGS;
#define DLI_OPT_ARGS_SIZE sizeof(DLI_OPT_ARGS)
```

Figure 4–3: “C” Definition of DLI Optional Arguments Structure

Freeway Header	ICP Header	Protocol Header	Data

Figure 4–4: Freeway DLI Data Format

Table 4–5: DLI Protocol-Specific Optional Arguments Data Structure

Field	Description
usFWPacketType	This field contains the type of Freeway data packet, either FW_CONTROL or FW_DATA. Your application must fill this field correctly if it uses *Raw* operation.
usFWCommand	This field contains the command that your application wishes to send to the Freeway server to request its services. These services include local session management as well as message multiplexing between client applications and the ICP protocol service. Your application can communicate directly with the Freeway services through this command field using *Raw* operation. Your application can use all commands that are supported by the Freeway server, except the FW_OPEN_SESS_CMD, and FW_CLOSE_SESS_CMD commands. Regardless of the type of operation, *Raw* or *Normal*, the DLI rejects these commands if they are issued to the Freeway server. If you wish to use these two commands, your application must bypass DLI services and use TSI services directly. Likewise, DLI returns packets received from the Freeway server in any combination of usFWPacketType and usFWCommand, with one exception: it does not return a packet that has usFWPacketType equal to FW_CONTROL, and usFWCommand equal to FW_OPEN_SESS_RSP or FW_CLOSE_SESS_RSP. Valid commands are: FW_ICP_WRITE, FW_ICP_WRITE_EXP, FW_ICP_READ, FW_GET_TIME_CMD, FW_SET_TIME_CMD, and FW_GET_VERSION_CMD. Also refer to the *Freeway Client-Server Interface Control Document*.
usFWStatus	This field contains the status of a request that was received and processed by the Freeway server. This field should be set to a value agreed upon between your application and the Freeway server. When your application issues a write request to the Freeway server, it should zero out the usFWStatus field. On return from the Freeway server, usFWStatus might contain useful information related to your request. Refer to the *Freeway Client-Server Interface Control Document*.
usICPClientID	This field (formerly called su_id) is provided for compatibility. This field is used exclusively by the X.25 protocol service.

Table 4–5: DLI Protocol-Specific Optional Arguments Data Structure *(Cont'd)*

Field	Description
usICPServerID	This field (formerly called sp_id) is provided for compatibility. This field is used exclusively by the X.25 protocol service.
usICPCommand	This field contains the protocol-specific command that your application wishes to send to the protocol service on the ICP. Your application can issue any commands that are supported by the protocol service.
iICPStatus	This field contains the status of a request that was received and processed by the ICP protocol service
usICPParms	This field contains additional protocol-specific parameters.
usProtCommand	This field contains the protocol-specific command that your application wishes to send to the ICP protocol service.
iProtModifier	This field contains the protocol-specific subcommand that your application wishes to send to the ICP protocol service.
usProtLinkID	This field contains the protocol-specific link ID.
usProtCircuitID	This field contains the protocol-specific circuit ID.
usProtSessionID	This field contains the protocol-specific session ID.
usProtSequence	This field contains the protocol-specific sequence ID.
usProtXParms	This field contains additional protocol-specific session parameters.

4.2 dlBufAlloc

The dlBufAlloc function allocates a ***fixed-size buffer*** that is maintained by DLI services. Buffers obtained through dlBufAlloc are normally used for data transmission; however, your application can use them for other purposes. To avoid a buffer depletion problem, your application must return all unused buffers to DLI using dlBufFree (Section 4.3).

Though you are not required to use dlBufAlloc, you should consider using it for all DLI I/O operations for the following reasons:

- DLI uses TSI buffer services and handles all buffer overhead requirements
- DLI allocates all buffers up front, resulting in better real-time performance than the normal C malloc and free functions
- The number of buffers is configurable for operating environments with limited system resources (maxBuffers TSI parameter, page 148)

The DLI requires appropriate headers that are prefixed to the data to be transmitted to the Freeway server. To enhance performance, the DLI implementation uses the memory area just before the data area to store its headers. Due to this implementation, if your application does not wish to use the DLI buffer management, it must allocate memory that contains not only its data but also the DLI headers. To obtain the amount of memory required for the DLI overhead, your application can call dlPoll with the DLI_POLL_GET_SYS_CFG option (usOverhead field on page 79). Refer to Section 2.4 on page 40 for information on buffer management issues.

Synopsis

```
char *dlBufAlloc (
    int          iBufLen );         /* Minimum size required                    */
```

Parameters

int iBufLen The length of the buffer to be allocated by DLI. It must not be larger than the maximum buffer size value. After calling dlInit, call dlPoll using the DLI_POLL_GET_SYS_CFG option to obtain the maximum buffer size allowed by the DLI (iMaxBufSize field of the DLI_SYS_CFG structure, page 79).

Note

Currently the DLI buffer pool is built with buffers all having the same length, so each call to dlBufAlloc yields a buffer of the length specified in the iMaxBufSize field of the DLI_SYS_CFG structure (page 79), regardless of the value of iBufLen.

Returns

If the dlBufAlloc function completes successfully, it returns the address of the buffer data area to be used by your application. Immediately preceding the buffer (at lower-numbered memory addresses than the buffer address) are headers that are manipulated by DLI (refer back to Figure 2–5 on page 48). These areas must not be modified by the application. If an error occurs, dlBufAlloc returns NULL, and dlerrno contains one of the following error codes (listed alphabetically):

DLI_BUFA_ERR_NEVER_INIT DLI was never initialized (dlInit).

Action: Review your application and try again.

DLI_BUFA_ERR_NO_BUFS DLI exhausted buffers.

Action: Severe error; consider increasing the number of buffers in the TSI configuration services. Review your application and ensure it releases unused buffers to DLI.

DLI_BUFA_ERR_SIZE_EXCEEDED iSize value is too large.

Action: Use dlPoll to get the maximum buffer size allowed or consider your configuration file.

For additional error codes, refer to Appendix B.

4.3 dlBufFree

Your application must use dlBufFree to release a DLI buffer that it allocated using dlBufAlloc. It must also release any read buffer that DLI allocated in dlRead (Section 4.12). The buffer is returned to the DLI internal free buffer pool. It is the responsibility of your application to prevent buffer depletion problems by releasing the unused DLI buffers.

Synopsis

```
char *dlBufFree (
        char        *pBuf );            /* Buffer to return to buffer pool          */
```

Parameters

char *pBuf   The address of the DLI buffer that was allocated by dlBufAlloc.

Returns

If the dlBufFree function completes successfully, it returns the value of pBuf. Otherwise it returns NULL, and dlerrno contains one of the following error codes (listed alphabetically):

DLI_BUFF_ERR_INVALID_BUF   Your application requested DLI to free a buffer that points to NULL.

Action: Revise your application logic, and try again.

DLI_BUFF_ERR_NEVER_INIT   DLI was never initialized (dlInit).

Action: Review your application and try again.

DLI_BUFF_ERR_TSI_FREE_ERR   DLI called tBufFree to free a buffer and TSI returned an error.

Action: Review TSI error codes, review your application and try again.

For additional error codes, refer to Appendix B.

4.4 dlClose

In *Normal* operation, the dlClose function terminates an active session between your application, the Freeway server, and the remote data link application. If dlOpen was invoked for *Raw* operation, dlClose terminates a session only with the Freeway server for this session. The underlying transport connection is also disconnected. When using non-blocking I/O, your application should call dlPoll to cancel all outstanding I/O requests before it issues the dlClose call.

All resources associated with a session are released with a dlClose request. If you observe "…INVALID_STATE" errors in the DLI and TSI log files with close requests, these may be normal since close processing is forced to completion when some types of abnormal conditions are recognized by DLI/TSI.

Synopsis

```
int dlClose (
    int     iSessionID,       /* Session ID from dlOpen              */
    int     iCloseMode );     /* Close mode (normal or force)        */
```

Parameters

int iSessionID   The session ID returned by the dlOpen or dlListen function call.

int iCloseMode   This parameter allows your application to request DLI to terminate an active session in the following close modes:

DLI_CLOSE_FORCE   When your application issues a force close for an active session, DLI empties the I/O queues and proceeds with the session termination process without considering the status of I/O queues. Note that when your application issues a dlTerm while active sessions exist, DLI itself issues a force close request before it frees the DLI service structure.

DLI_CLOSE_NORMAL   DLI rejects a normal close request if its internal input and output queues contain outstanding I/O requests. If either queue is not empty, DLI rejects the normal close request. To successfully issue a normal close request for an active session, your application must first empty the I/O queues using dlPoll calls.

Returns

If the dlClose function completes successfully, it returns OK. Otherwise, it returns ERROR, and dlerrno contains one of the following error codes (listed alphabetically):

DLI_EWOULDBLOCK The session was configured for non-blocking I/O, and could not be closed immediately.

Action: Use dlPoll to check if your request completed. You might wish to program your application to be awakened by your own interrupt service routine that you provided when you called the dlInit, dlOpen, or dlListen function. Refer to Section 2.2 on page 32 for information on non-blocking I/O.

DLI_CLOS_ERR_FW_INVALID_RSP DLI encountered an invalid response from the Freeway server.

Action: Review your trace file and verify the Freeway version.

DLI_CLOS_ERR_FW_INVALID_SESS Freeway did not recognize the session ID provided by DLI on the close session request.

Action: Check your application’s logic, and evaluate the DLI trace and error logs.

DLI_CLOS_ERR_FW_QADD_FAILED DLI failed to access its internal I/O queues.

Action: Severe error; terminate your application and try again.

DLI_CLOS_ERR_FW_TOO_MANY_ERRORS DLI encountered too many I/O error conditions while it attempted to close this session.

Action: Review your operating environment and your DLI session configuration.

DLI_CLOS_ERR_FW_UNK_STATUS Freeway's returned status is not recognized by DLI.

Action: Verify the versions of your Freeway and DLI services.

DLI_CLOS_ERR_ICP_INVALID_RSP DLI encountered an invalid response from the ICP.

Action: Verify the ICP version.

DLI_CLOS_ERR_ICP_INVALID_STATUS The ICP returned status is not recognized by DLI.

Action: Verify the versions of your Freeway, ICP, and DLI services.

DLI_CLOS_ERR_ICP_QADD_FAILED DLI failed to access its internal I/O queues.

Action: Severe error; terminate your application and try again.

DLI_CLOS_ERR_ICP_TOO_MANY_ERRORS DLI encountered too many I/O error conditions while it attempted to close this session.

Action: Review your operating environment and your DLI session configuration.

DLI_CLOS_ERR_INVALID_MODE Invalid mode for close request (use DLI_CLOSE_NORMAL or DLI_CLOSE_FORCE).

Action: Review your application logic.

DLI_CLOS_ERR_INVALID_SESSID The provided session ID is invalid.

Action: Review your application logic.

DLI_CLOS_ERR_INVALID_STATE DLI encountered an invalid state in its state processing machine.

Action: Review the DLI trace and error logs.

DLI_CLOS_ERR_LINK_INVALID_RSP The protocol service's returned response is not recognized by DLI.

Action: Verify the versions of your Freeway, ICP, protocol service, and DLI services.

DLI_CLOS_ERR_LINK_INVALID_STATUS The ICP returned status is not recognized by DLI.

Action: Verify the versions of your Freeway, ICP, and DLI services.

DLI_CLOS_ERR_LINK_QADD_FAILED DLI failed to access its internal I/O queues.

Action: Severe error; terminate your application and try again.

DLI_CLOS_ERR_LINK_TOO_MANY_ERRORS DLI encountered too many I/O error conditions while it attempted to close this session.

Action: Review your operating environment and your DLI session configuration.

DLI_CLOS_ERR_NEVER_INIT DLI was never initialized. You must call dlInit before using this function.

Action: Correct your application and try again.

DLI_CLOS_ERR_Q_NOT_EMPTY Your application requested close on a given session, and the internal I/O queues for that session are not empty.

Action: Review your application and try again. Consider using dlClose with the DLI_CLOSE_FORCE option.

DLI_CLOS_ERR_TOO_MANY_ERRORS DLI encountered too many I/O error conditions while it attempted to close this session.

Action: Review your operating environment and your DLI session configuration.

For additional error codes, refer to Appendix B.

4.5 dlControl

The dlControl function sends control messages to Freeway. Currently, the only control message supported is DLI_CTRL_RESET_ICP which resets an ICP and downloads the protocol software from the boot server. Refer to Section 5.4 on page 179 for example dlControl code and application program detection of ICP reset.

When an ICP download is requested, each client connected to the affected ICP receives a Freeway control packet with the usFWStatus field set to FW_ICP_DWNLD_ACTIVE. At this point the application should not issue any further dlWrite requests.

To detect that an ICP reset is complete, an application must issue dlRead requests (see Section 4.12) using optional arguments. It must then examine the usFWPacketType, usFWCommand, and usFWStatus fields (refer to the DLI protocol-specific optional arguments in Section 4.1.3.3 on page 83).

In a control packet, the usFWPacketType field will be set to FW_CONTROL. Freeway control packets do not contain the ICP or Protocol header fields. The usFWCommand field will be set to FW_ICP_STATUS_RSP for an ICP reset/download packet.

When the ICP download completes, each client connected to the affected ICP receives a Freeway control packet with the usFWStatus field set to FW_ICP_DWNLD_OK. At this time the application should cancel all pending reads and writes and call dlClose to terminate the session. After the session is closed, it can be reopened and reused.

Synopsis

```
int dlControl (
        char    *cSessionName,      /* Session name in DLI config file      */
        int     iCommand,           /* Control command                      */
        int     (*fUsrIOCH) (char *pUsrCB, int iSessionID) );
                                    /* Optional IOCH                        */
```

Parameters

char *cSessionName A string of characters that specifies the name of the desired session definition entry in the DLI binary configuration file. The associated configuration entry defines the characteristics of the data link session on the ICP you are about to reset.

int iCommand Valid values: DLI_CTRL_RESET_ICP

int (*fUsrIOCH) (char *pUsrCB, int iSessionID) The optional address of the IOCH function that you wish DLI to invoke immediately after it services a non-blocking I/O condition *for this session* as specified by iSessionID. You must write this function yourself. The DLI passes the pUsrCB value (that you provided with the dlInit function) and the session ID of the session that receives the I/O condition notification.

Returns

If the dlControl function completes successfully, it returns OK. Otherwise, it returns ERROR, and dlerrno contains one of the following error codes (listed alphabetically). Also see dlOpen in Section 4.8 on page 106 for a list of possible dlControl error returns.

DLI_CTRL_ERR_FAILED DLI failed to open a control session with a remote data link application.

Action: Review your session configuration parameters. You can review DLI trace and error log for additional information.

DLI_CTRL_ERR_FW_FTP_FAIL DLI failed to ftp the ICP code to Freeway.

Action: Verify that the ICP code exists and its path is valid.

DLI_CTRL_ERR_FW_ICP_FAIL DLI failed to download the ICP code to an ICP.

Action: Verify that a valid ICP code is used.

DLI_CTRL_ERR_FW_INVALID_ICP Freeway encountered a non-existent ICP.

Action: Check your session configuration and Freeway hardware configuration.

DLI_CTRL_ERR_FW_INVALID_RSP DLI encountered an invalid response from the Freeway server.

Action: Verify the DLI and Freeway software versions.

DLI_CTRL_ERR_FW_INVALID_TYPE DLI encountered an invalid type from the Freeway response packet.

Action: Check the Freeway session configuration and its operational status.

DLI_CTRL_ERR_FW_SCRIPT_ERR DLI encountered an invalid ICP script.

Action: Verify that a valid ICP script is used, that the download script exists, and that its path is valid.

DLI_CTRL_ERR_FW_UNK_STATUS Freeway's returned status is unknown to DLI.

Action: Verify the DLI and Freeway software versions.

DLI_CTRL_ERR_INIT_FAILED The dlControl function failed to initialize itself through dlInit.

Action: Check your binary configuration file. If the default binary configuration file (dlicfg.bin) was used by DLI, verify its existence.

DLI_CTRL_ERR_INVALID_STATE The dlControl function encountered an invalid state in its state processing machine.

Action: Review the DLI trace and error logs.

DLI_CTRL_ERR_SESS_INIT_FAILED DLI failed to initialize the session entry for this control request.

Action: Check the DLI error log for additional error messages.

DLI_CTRL_ERR_TOO_MANY_ERRORS DLI encountered too many I/O error conditions that exceeded the maxErrors DLI parameter value specified for this session.

Action: Review your operating environment and your DLI session configuration.

For additional error codes, refer to Appendix B.

4.6 dlInit

The dlInit function is the first DLI function your application calls. It initializes the DLI services based upon the user's binary configuration file (described in Chapter 3) provided through the first parameter, cCfgFile.

Your application must call dlInit to ensure proper operation of the Freeway server.

Synopsis

```
int dlInit (
        char     *cCfgFile,            /* DLI binary configuration file name      */
        char     *pUsrCB,              /* I/O complete control block              */
        int      (*fUsrIOCH) (char *pUsrCB)); /* Optional IOCH                    */
```

Parameters

char *cCfgFile The DLI binary configuration file that contains all DLI run-time parameters as well as data link session parameters. This file results from execution of the DLI configuration preprocessor program, dlicfg. If this parameter is NULL, the default file (dliCfg.bin) is used. Whether or not you supply the configuration file name, the binary configuration file must exist in order for DLI to operate. An optional on-line configuration method is described in Section 3.3.4 on page 61.

char *pUsrCB The address of a user-defined control block that DLI passes as the first parameter to your supplied I/O completion handler (IOCH); see the fUsrIOCH parameter below. The DLI does not examine or change the contents of this structure. For blocking I/O, this parameter should be NULL.

int (*fUsrIOCH) (pchar *pUsrCB) The optional address of your general-purpose IOCH function that DLI invokes immediately after it services any I/O condition when DLI is configured for non-blocking I/O. This IOCH is called for any session that DLI is currently managing. You must write this function yourself. If your application uses blocking I/O, or you do not wish DLI to invoke an IOCH function, this parameter should be NULL. Either the dlInit or the dlOpen function can be used to supply the IOCH; however, the dlOpen IOCH requires a session ID, and is called for that particular session only.

Returns

The dlInit function returns immediately because it does not involve any I/O operations. If dlInit completes successfully, it returns OK. Otherwise it returns ERROR, and dlerrno contains one of the following error codes (listed alphabetically):

DLI_INIT_ERR_ACT_ADD_REM_FAILED DLI failed during the initialization process.

Action: Severe error; terminate your application and try again.

DLI_INIT_ERR_ACT_QINIT_FAILED DLI failed to initialize its internal active session queue.

Action: Check your system resources. Refer to Section 2.5 on page 53 to calculate system resources required by DLI and TSI.

DLI_INIT_ERR_ALREADY_INIT Your application already issued dlInit before. It either proceeds with DLI services or calls dlTerm before it can call this function again.

Action: Review your application and try again.

DLI_INIT_ERR_CFG_LOAD_FAILED DLI failed to load the system configuration parameters from the provided binary configuration file.

Action: Check the binary configuration file used by DLI. If your application calls this function directly, make sure the binary configuration file containing the configuration your application provides exists. If your application does not call this function directly, DLI calls this function for you; make sure the default configuration file (dliCfg.bin) exists. Review your application and try again.

DLI_INIT_ERR_DLICB_ALLOC_FAILED DLI failed to allocate memory for its internal system control block.

Action: Check your system resources. Refer to Section 2.5 on page 53 to calculate system resources required by DLI and TSI.

DLI_INIT_ERR_GET_TSI_CFG_FAILED DLI's request for TSI status failed.

Action: Check your TSI services, terminate your application and try again.

DLI_INIT_ERR_LOG_INIT_FAILED DLI failed to initialize its internal logging and tracing facility.

Action: Check your logging and tracing related parameters in the currently used DLI configuration file.

DLI_INIT_ERR_NAME_TOO_LONG The DLI configuration file name is too long.

Action: Reduce the DLI configuration file name length.

DLI_INIT_ERR_NO_RESOURCE No memory resource is available for DLI to start its services.

Action: Make sure your operating environment provides sufficient memory resources for your application. Refer to Section 2.5 on page 53 for more details.

DLI_INIT_ERR_NO_TRACE_BUF DLI is unable to allocate memory for the requested trace buffer.

Action: Review your configuration parameters and your system resources. If necessary, reduce the value of the traceSize parameter (page 63).

DLI_INIT_ERR_TASK_VAR_FAILED DLI failed to add its control block to the task variable list within VxWorks.

Action: This error occurs only with server-resident applications on VxWorks. Check your VxWorks system configuration.

DLI_INIT_ERR_TEXT_OPEN_FAILED DLI failed to open the DLI text configuration file.

Action: Check the supplied configuration file name. If a binary file is supplied, verify the name contains a '.' character. If a text file is supplied, verify the file name and its existence in the current directory (where the application program is executing)

DLI_INIT_ERR_TSI_INIT_FAILED DLI failed to initialize TSI services.

Action: Check your TSI configuration services, terminate your application and try again.

For additional error codes, refer to Appendix B.

4.7 dlListen

The dlListen function waits for a connection establishment from a remote data link application. This function does not apply to all data link protocols. For those protocols that do not have "listening" capability, use the dlOpen function with *Normal* operation.

Unlike dlOpen, dlListen does not allow *Raw* operation. The dlListen function is similar to the dlOpen function, except that it waits for a connection request to arrive instead of sending a connection request to a predefined destination. You should give special consideration in configuring the session timeout value, because your application might have to wait a long period of time before receiving any connection requests from remote data link applications.

Synopsis

```
int dlListen (
        char     *cSessionName,      /* Session name in DLI config file          */
        int      (*fUsrIOCH) (char *pUsrCB, int iSessionID) );
                                     /* Optional IOCH                            */
```

Parameters

char *cSessionName   A string of characters that specifies the name of the desired session definition entry in the DLI binary configuration file. The associated configuration entry defines the characteristics of the data link session your application wishes to connect with when the connection request arrives.

int (*fUsrIOCH) (char *pUsrCB, int iSessionID)   The optional address of the IOCH function that you wish DLI to invoke immediately after it services a non-blocking I/O condition *for this session* as specified by iSessionID. You must write this function yourself. The DLI passes the pUsrCB value (that you provided with the dlInit function) and the session ID of the session that receives the I/O condition notification. You can provide a different fUsrIOCH for each dlListen call, or you can use the same fUsrIOCH in multiple dlListen calls. If your application uses blocking I/O, or you do not wish DLI to invoke an IOCH function, this parameter should be

NULL. The dlInit function can also be used to define a general-purpose IOCH that is not restricted to one particular session. If the IOCH is given as a parameter in the dlListen call, that IOCH will be invoked when the session is either successfully established or has failed.

Returns

The dlListen function returns a non-negative session ID if it successfully connects to the remote data link application or if it is in the process of connecting your application to the remote application. This ID uniquely identifies a DLI session between your application and the remote application. The session ID has a value between zero and the maximum number of sessions (DLI maxSess configuration parameter on page 63) minus 1.

If your application is configured for non-blocking I/O, it must either use dlerrno or call dlPoll with the DLI_POLL_GET_SESS_STATUS option to determine the status of the connection. If your application is configured for blocking I/O, the returned session ID indicates a successful connection.

If this function returns ERROR, the connection failed, and dlerrno contains one of the following error codes (listed alphabetically). Also see the DLI_OPEN error return codes on page 109.

DLI_EWOULDBLOCK   This value is set only if the return value is a valid session ID. The session was configured for non-blocking I/O, and a connection was not established immediately.

Action: Use dlPoll to check if your request completed. You might wish to program your application to be awakened by your own IOCH that you provided when you called the dlInit function or this function. Refer to Section 2.2 on page 32 for information on non-blocking I/O.

DLI_LSTN_ERR_INIT_FAILED DLI failed to initialize its services. This error occurs only if your application did not explicitly call the dlInit function.

Action: Check your DLI binary configuration file. If the default file (dliCfg.bin) was used by DLI, verify its existence.

DLI_LSTN_ERR_INVALID_STATE DLI encountered an invalid state in its state processing machine.

Action: Review the DLI trace and error logs.

DLI_LSTN_ERR_SESS_INIT_FAILED DLI failed to initialize the session entry for this listen request.

Action: Check the DLI error log for additional error messages.

For additional error codes, refer to the dlOpen function (Section 4.8) and Appendix B.

4.8 dlOpen

In *Normal* operation, the dlOpen function establishes a data link session with a remote application. If dlOpen is invoked for *Raw* operation (that is, the protocol DLI configuration parameter on page 65 is set to "raw"), it establishes a connection to the Freeway server only. However, there are occasions in *Normal* operation that require a *Raw* write or read request; refer to Section 2.3 on page 34 and to the dlRead and dlWrite functions for more information about *Raw* operation.

The dlOpen function configures the ICP link only if there are protocol-specific parameters in the session definition and the cfgLink DLI configuration parameter (page 64) is set to "yes," which is the default. The dlOpen function enables the link only if the enable parameter (page 64) is set to "yes," which is the default.

Note

If you need to request session status to obtain the maximum buffer size (which may change due to negotiation procedures during dlOpen), your application should wait until after a successful dlOpen before calling dlPoll with the DLI_POLL_GET_SESS_STATUS option (Section 4.1.3.2 on page 80). See Section 2.4.2.3 on page 49 for details of the negotiation process.

Caution

It is critical for the client application to receive the dlOpen completion status before making any other DLI requests; otherwise, subsequent requests will fail. After the dlOpen completion, however, you do not have to maintain a one-to-one correspondence between DLI requests and dlRead calls.

Synopsis

```
int dlOpen (
        char      *cSessionName,     /* Session name in DLI config file      */
        int       (*fUsrIOCH) (char *pUsrCB, int iSessionID) );
                                     /* Optional IOCH                        */
```

Parameters

char *cSessionName A string of characters that specifies the name of the desired session definition entry in the DLI binary configuration file. The associated configuration entry defines the characteristics of the data link session you are about to open. However, in *Raw* operation all data link related parameters that are defined for this session are ignored.

int (*fUsrIOCH) (char *pUsrCB, int iSessionID) The optional address of the IOCH function that you wish DLI to invoke immediately after it services a non-blocking I/O condition *for this session* as specified by iSessionID. You must write this function yourself. The DLI passes the pUsrCB value (that you provided with the dlInit function) and the session ID of the session that receives the I/O condition notification. You can provide a different fUsrIOCH for each dlOpen call, or you can use the same fUsrIOCH in multiple dlOpen calls. If your application uses blocking I/O, or you do not wish DLI to invoke an IOCH function, this parameter should be NULL. The dlInit function can also be used to define a general-purpose IOCH that is not restricted to one particular session. If the IOCH is given as a parameter in the dlOpen call, that IOCH will be invoked when the session is either successfully established or has failed.

Returns

The dlOpen function returns a non-negative session ID if it successfully connects to the remote data link application or if it is in the process of connecting your application to the remote application. This ID uniquely identifies a DLI session between your application and the remote application. The session ID has a value between zero and the maximum number of sessions (maxSess DLI configuration parameter on page 63) minus 1.

If your application is configured for non-blocking I/O, it must either use dlerrno or call dlPoll with the DLI_POLL_GET_SESS_STATUS option to determine the status of the connection. If your application is configured for blocking I/O, the returned session ID indicates a successful connection.

If this function returns ERROR, the connection failed, and your application must check dlerrno which contains one of the following error codes (listed alphabetically):

DLI_EWOULDBLOCK   This value is set only if the return value is a valid session ID. The session was configured for non-blocking I/O, and could not be established immediately.

Action: Use dlPoll to check if your request completed. You might wish to program your application to be awakened by your own IOCH that you provided when you called the dlInit function or this function. Refer to Section 2.2 on page 32 for information on non-blocking I/O.

DLI_OPEN_ERR_CFG_INVALID_RSP   DLI encountered an invalid response from the ICP when it attempted to configure the link before activating it.

Action: Review the DLI trace, verify the ICP software version, and try again. If your application is using *Raw* operation, review the sequence and specific commands being sent to the ICP.

DLI_OPEN_ERR_CFG_INVALID_STATUS   DLI encountered an invalid status from the ICP when it attempted to configure the link. This often indicates that the ICP rejected the configuration command from DLI.

Action: Review the DLI trace, verify the ICP software version, and try again. If your application is using *Raw* operation, review the sequence and specific commands being sent to the ICP.

DLI_OPEN_ERR_CFG_QADD_FAILED DLI failed to access the internal I/O queues while attempting to configure the link.

Action: Severe error; terminate your application and try again.

DLI_OPEN_ERR_CFG_TOO_MANY_ERRORS DLI encountered too many I/O error conditions that exceeded the maxErrors DLI parameter value specified for this session (page 64).

Action: Review your operating environment and your DLI session configuration.

DLI_OPEN_ERR_FAILED DLI failed to open a session with a remote data link application.

Action: Review your session configuration parameters. You can review the DLI trace and error log for additional information.

DLI_OPEN_ERR_FW_ICP_NOT_OP Freeway encountered a non-operational ICP.

Action: Check your session configuration and Freeway operational status.

DLI_OPEN_ERR_FW_INVALID_COMMAND DLI encountered an invalid command in the Freeway response packet.

Action: Verify the DLI and Freeway software versions.

DLI_OPEN_ERR_FW_INVALID_ICP Freeway encountered a non-existent ICP.

Action: Check your session configuration and Freeway hardware configuration.

DLI_OPEN_ERR_FW_INVALID_RSP DLI encountered an invalid response from the Freeway server.

Action: Verify the DLI and Freeway software versions.

DLI_OPEN_ERR_FW_INVALID_TYPE DLI encountered an invalid type from the Freeway response packet.

Action: Verify the DLI and Freeway software versions.

DLI_OPEN_ERR_FW_NO_SESS Freeway failed to create an additional session for your application.

Action: Check the Freeway session configuration and its operational status.

DLI_OPEN_ERR_FW_QADD_FAILED DLI failed to access the internal I/O queues.

Action: Severe error; terminate your application and try again.

DLI_OPEN_ERR_FW_TOO_MANY_ERRORS DLI encountered too many I/O error conditions that exceeded the maxErrors DLI parameter value specified for this session (page 64).

Action: Review your operating environment and your DLI session configuration.

DLI_OPEN_ERR_FW_UNK_STATUS Freeway's returned status is unknown to DLI.

Action: Verify the DLI and Freeway software versions.

DLI_OPEN_ERR_ICP_INVALID_RSP DLI encountered an invalid response from the ICP protocol service.

Action: Verify the versions of the Freeway, ICP, and DLI services. This error occurs only in *Normal* operation.

DLI_OPEN_ERR_ICP_INVALID_STATUS The ICP returned status is unknown to DLI.

Action: Verify the versions of your Freeway, ICP, and DLI services. This error occurs only in *Normal* operation.

DLI_OPEN_ERR_ICP_QADD_FAILED DLI failed to access its internal I/O queues while it attempted to connect to the specified ICP.

Action: Severe error; terminate your application and try again. This error occurs only in *Normal* operation.

DLI_OPEN_ERR_ICP_TOO_MANY_ERRORS DLI encountered too many I/O error conditions that exceeded the maxErrors DLI parameter value specified for this session (page 64).

Action: Review your operating environment and your DLI session configuration.

DLI_OPEN_ERR_INIT_FAILED DLI failed to initialize its services. This error occurs only if your application does not explicitly call dlInit function.

Action: Check your binary configuration file. If the default binary configuration file (dliCfg.bin) was used by DLI, verify its existence.

DLI_OPEN_ERR_INVALID_STATE DLI encountered an invalid state in its state processing machine.

Action: Review the DLI trace and error logs.

DLI_OPEN_ERR_LINK_INVALID_RSP DLI encountered an invalid response from the ICP protocol service.

Action: Verify the versions of your Freeway, ICP, ICP protocol services, and DLI services. This error occurs only with *Normal* operation.

DLI_OPEN_ERR_LINK_INVALID_STATUS ICP protocol service's returned status is unknown to DLI.

Action: Verify versions of your Freeway, ICP, and your DLI services. This error occurs only in *Normal* operation.

DLI_OPEN_ERR_LINK_QADD_FAILED DLI failed to access its internal I/O queues while it attempted to connect to the remote data link application.

Action: Severe error; terminate your application and try again. This error occurs only in *Normal* operation.

DLI_OPEN_ERR_LINK_TOO_MANY_ERRORS DLI encountered too many I/O error conditions that exceeded the maxErrors DLI parameter value specified for this session (page 64).

Action: Review your operating environment and your DLI session configuration.

DLI_OPEN_ERR_SESS_INIT_FAILED DLI failed to initialize the session entry for this open request.

Action: Check the DLI error log for additional error messages.

DLI_OPEN_ERR_TOO_MANY_ERRORS DLI encountered too many I/O error conditions that exceeded the maxErrors DLI parameter value specified for this session (page 64).

Action: Review your operating environment and your DLI session configuration.

For additional error codes, refer to Appendix B.

4.9 dlpErrString

The dlpErrString function allows the user to print the text message associated with a DLI error defined by the user input (a valid dlerrno value). This function can be invoked without DLI initialization. The function returns a pointer to the textual description of the DLI error number supplied with the function call.

Synopsis

```
char  *dlpErrString (
          int        dlErrNo);       /* DLI error number (a valid dlerrno value)       */
```

Parameters

int dlErrNo DLI error number of the associated text description (must be a valid dlerrno value).

Returns

If the dlpErrString function completes successfully, it returns a pointer to a NULL-terminated character string associated with the DLI error number (dlErrNo) supplied in the function call.

If this function completes unsuccessfully, it returns NULL. The dlpErrString function does not change the current dlerrno value, so that it will still reflect the global dlerrno value at the time dlpErrString was invoked. If DLI logging is enabled, the following message is logged:

DLI_PRTSTRG_ERR_UNKNOWN_ERROR_NBR

For additional error codes, refer to Appendix B.

4.10 dlPoll

The dlPoll function queries either DLI general information or session-related status or configuration information. When using *Raw* operation, dlPoll can be used to query I/O completion status. Your application can call this function as often as it needs. This function does not involve any I/O operations.

Synopsis

```
int dlPoll (
    int         iSessionID,         /* Session ID                           */
    int         iPollType,          /* Request type                         */
    char        **ppBuf,            /* Poll-type dependent parameter        */
    int         *piBufLen,          /* Size of I/O buffer in bytes          */
    char        *pStat              /* Status or configuration buffer       */
    DLI_OPT_ARGS **ppOptArgs); /* Protocol optional arguments               */
```

Parameters

int iSessionID   This session ID uniquely identifies an active session serviced by the DLI. This ID is returned from a dlOpen or dlListen function.

int iPollType   The type of poll request to the DLI. Valid poll types are:

DLI_POLL_GET_CFG_LIST   Request DLI to get a list of all DLI session definition names. The list is returned through the ppBuf parameter in NULL-terminated string lists. The piBufLen parameter indicates the number of session definition names in the list. The list does not contain the definition of the "main" section.

DLI_POLL_GET_SESS_STATUS   Request DLI to get the current session status. The session status is returned through the DLI_SESS_STAT structure. The pointer to the DLI_SESS_STAT structure (Section 4.1.3.2 on page 80) must be provided through the pStat parameter. This option should be used sparingly because it checks the entire input and output queues for I/O completion status.

Caution

When using blocking I/O, the iQNumReadDone field of the DLI_SESS_STAT structure is always zero and must not be used to determine when to queue a dlRead request.

DLI_POLL_GET_SYS_CFG Call dlPoll after the dlInit call to request DLI to get the DLI system configuration. Set the iSessionID parameter to zero. The system status is returned through the DLI_SYS_CFG structure (Section 4.1.3.1 on page 78). The pointer to the DLI_SYS_CFG structure must be provided through the pStat parameter.

DLI_POLL_READ_CANCEL Request DLI to remove the first read request in its input queue regardless of the completion status of the read request. If the content of the ppBuf (*ppBuf) parameter is NULL, the DLI removes the first entry in the input queue regardless of its completion status. If the content of ppBuf is not NULL, the DLI searches through its input queue for a matching address pointer. If it finds a match, it removes that request regardless of its completion status. When using non-blocking I/O, your application should call dlPoll to cancel all outstanding I/O requests before it issues the dlClose call.

DLI_POLL_READ_COMPLETE Request DLI to remove the first read request from this session's input queue that is either complete, timed-out, or a read error has occurred. The address and length of the buffer are returned through the ppBuf and piBufLen parameters. This request removes the first entry in the input queue if and only if the entry is marked "read complete," "read timed-out," or "read error." In all cases, the application is responsible for freeing the returned buffer.

DLI_POLL_WRITE_CANCEL Request DLI to remove the first write request in its output queue regardless of the completion status of the write request. If the content of the ppBuf (*ppBuf) parameter is NULL, the DLI removes the first entry in the output queue regardless of its completion status. If the content of ppBuf is *not* NULL, the DLI searches through its output queue for a matching address pointer. If it finds a match, it removes that request regardless of its completion status.

DLI_POLL_WRITE_COMPLETE Request DLI to remove the first write request in its output queue that is either complete or timed-out. This request removes the first entry in the output queue if and only if the request is marked “write complete.”

DLI_POLL_TRACE_ON Turn on the DLI/TSI tracing facility. If trace is configured in the DLI configuration file (traceName and traceSize parameters on page 63), the tracing facility is automatically *on*. Your application can turn tracing on or off as often as it needs.

DLI_POLL_TRACE_OFF Turn off the DLI/TSI tracing facility. If trace is not configured in the DLI and TSI configuration files, this has no affect.

DLI_POLL_TRACE_STORE Write your own information into the DLI trace buffer. Use pStat to indicate the area of memory to be copied to the trace buffer, and iBufLen to indicate the length of the area to be copied. The length of your trace area must be less than or equal to the size of the trace buffer (traceSize parameter on page 63). Otherwise, your trace area will be truncated when copied into the DLI trace buffer.

DLI_POLL_TRACE_WRITE Force the DLI/TSI to write the trace buffer into the trace file. The name of the trace file is defined by the traceNameDLI configuration parameter (page 63). Your application can force the trace buffer to be written to the trace file as often as it needs.

char *ppBuf This parameter specifies an address of a pointer to a buffer area. This parameter must not be NULL except when the poll type is to get session or system status.

int *piBufLen This field contains the length of the buffer pointed to by the content of the ppBuf parameter, or the number of entries in the configuration list also pointed to by the content of the ppBuf parameter. If a NULL value is passed, the dlPoll request is returned with the DLI_POLL_ERR_BUF_LEN_PTR_NULL error code.

char *pStat This field is the pointer to either the session status or the system configuration. If the request is for the session status, this field is the address of the DLI_SESS_STAT structure. Otherwise, it points to the DLI_SYS_CFG structure.

DLI_OPT_ARGS **ppOptArgs This field contains the address of a pointer to the optional arguments structure that was provided by the dlRead or dlWrite function.

Returns

If the dlPoll function completes successfully, it returns OK. Otherwise it returns ERROR, and dlerrno contains one of the following error codes (listed alphabetically):

DLI_POLL_ERR_BAD_PTR ppBuf pointer must not be NULL for this request.

Action: Review your application and try again.

DLI_POLL_ERR_BUF_LEN_PTR_NULL The piBufLen pointer must not be NULL for this request.

Action: Review your application and try again.

DLI_POLL_ERR_BUF_NOT_FOUND DLI could not cancel a read or a write request based on the buffer address provided by your application.

Action: Make sure that you provide a correct buffer address when you request a read or write cancellation.

DLI_POLL_ERR_GETLIST_FAILED DLI failed to get a list of session definition entries from the DLI configuration file.

Action: Verify the configuration file.

DLI_POLL_ERR_GET_TSI_CFG_FAILED DLI's request for TSI status failed.

Action: Check your TSI services, terminate your application and try again.

DLI_POLL_ERR_INVALID_IOQ DLI encountered an internal error with the I/O queue.

Action: Terminate your application and try again.

DLI_POLL_ERR_INVALID_REQ_TYPE Your application issued an invalid request type.

Action: Review your application. Verify your version of DLI.

DLI_POLL_ERR_INVALID_SESSID Your session is no longer valid.

Action: Review your error log, terminate your application and try again.

DLI_POLL_ERR_IO_FATAL A fatal I/O error occurred. The application can assume the connection has been terminated.

Action: Cancel all outstanding read/write requests and free the buffers. Close the session. After the session is closed, another session connection can be attempted.

DLI_POLL_ERR_NEVER_INIT DLI was never initialized.

Action: Revise your application and try again.

DLI_POLL_ERR_OVERFLOW Your current read request has overflowed the maximum buffer size. The buffer returned contains as much of the data as would fit, and the remainder has been discarded.

Action: Review your TSI configurations for both the client and Freeway. Ensure that the maxBufSize TSI configuration parameters (page 148) are defined as intended. Increase the buffer size supplied to dlRead up to the maximum defined for the session.

DLI_POLL_ERR_QEMPTY Your application issued a poll request on an I/O queue that is empty.

Action: Review your application.

DLI_POLL_ERR_QREM_FAILED DLI failed to remove an I/O request from one of the I/O queues.

Action: Severe error; terminate your application and try again.

DLI_POLL_ERR_READ_ERROR DLI encountered a severe error while reading data from TSI.

Action: Review the DLI and TSI trace files and error logs. Terminate your application and try again.

DLI_POLL_ERR_READ_NOT_COMPLETE There is no complete read request to return to the caller.

Action: Your application can check the request again at a later time.

DLI_POLL_ERR_READ_QREM_FAILED DLI failed to remove an I/O request from one of the I/O queues.

Action: Severe error; terminate your application and try again.

DLI_POLL_ERR_READ_TIMEOUT The return read request did not complete successfully due to timeout.

Action: Review your application. You might need to change the timeout parameter in your DLI configuration file.

DLI_POLL_ERR_UNBIND The connection supporting this session between the client application and Freeway has been closed. The system has performed “Unbind” processing. The connection was closed either because of a “Force Unbind” received from the peer entity (Freeway or client), or because of a failure with the I/O connection.

Action: Cancel all outstanding read/write requests and free the buffers. Close the session. After the session is closed, another session connection can be attempted. Examine the message logs on the peer system (if the error occurred in the client application, examine the Freeway log).

DLI_POLL_ERR_WRITE_ERROR The ICP could not send the data to the remote system.

Action: The global variable iICPStatus and the optional arguments iICPStatus field contain the ICP error number.

DLI_POLL_ERR_WRITE_NOT_COMPLETE There is no complete write request to return to the caller.

Action: Your application can check the request again at a later time.

DLI_POLL_ERR_WRITE_TIMEOUT The return write request did not complete successfully due to timeout.

Action: Review your application. Change the TSI timeout configuration parameter (page 149).

For additional error codes, refer to Appendix B.

4.11 dlPost

The dlPost function operates only in the VxWorks environment where the basic non-blocking I/O system services are not provided. It signals the I/O server task to begin processing I/O requests queued by your application.

Currently, dlPost implements a controlled task switch environment for VxWorks using binary semaphore mechanisms. Your application must call this function as the last operation before it relinquishes the task control to the operating system (i.e. tasking semaphore). Your application must take a special consideration in operating in a VxWorks environment. Refer to Appendix C for designing and implementing server-resident applications under a VxWorks environment.

Synopsis

```
int     dlPost ( void );
```

Parameters

None.

Returns

If the dlPost function completes successfully, it returns OK. Otherwise it returns ERROR, and dlerrno contains one of the following error codes (listed alphabetically):

DLI_POST_ERR_NEVER_INIT   The DLI was never initialized (dlInit).

Action: Correct your application and try again.

DLI_POST_ERR_TSI_POST_ERR   The TSI post function failed.

Action: Check for additional error codes reported by TSI.

For additional error codes, refer to Appendix B.

4.12 dlRead

Typically the dlRead function is used in *Normal* operation to interact with Freeway only for the purpose of receiving data from the remote data link application; in this case the optional arguments parameter is set to NULL. However, if you need to receive status or configuration information from Freeway or process protocol-specific incoming data, your application must issue a *Raw* dlRead by providing the optional arguments parameter for DLI to fill with the server and protocol specifics.

Synopsis

```
int dlRead (
        int           iSessionID,          /* Session ID from dlOpen               */
        char          **ppBuf,             /* Buffer to receive data               */
        int           iBufLen,             /* Maximum bytes to be returned         */
        DLI_OPT_ARGS *pOptArgs );          /* Optional arguments structure         */
```

Parameters

int iSessionID This session ID uniquely identifies an active session serviced by DLI. This ID is returned from the dlOpen or dlListen function call.

char **ppBuf The address of a pointer to a DLI read buffer. The buffer can be allocated using the dlBufAlloc function or by a similar C function. However, if the buffer is allocated by a function other than dlBufAlloc, your application must provide sufficient header space for the DLI to store its internal information related to this buffer (usOverhead field on page 79). This field must not be NULL; however, its content can be a NULL pointer. If its content is NULL, DLI allocates a buffer for your application. See Section 2.4 on page 40 for buffer management information.

If you let DLI allocate the read buffer, *your application is still responsible for releasing that buffer* when it no longer needs it, using the dlBufFree function. You should consider specifying the content of this field as a NULL pointer if your TSI connection is configured for shared memory. Refer to the *Freeway Transport Subsystem Interface Reference Guide* for more details on the shared memory transport protocol. If the session is using non-blocking I/O, your application must not reuse this buffer until the dlRead request is complete.

int iBufLen The maximum number of bytes of data to be read by DLI (excluding headers). If the actual data length is more than this value, DLI discards the extra data and returns a DLI_READ_ERR_OVERFLOW error indication to the application. After calling dlOpen, call dlPoll using the DLI_POLL_GET_SESS_STATUS option to obtain the maximum application data buffer size allowed by the DLI (usMaxSessBufSize field of the DLI_SESS_STAT structure, page 82).

DLI_OPT_ARGS *pOptArgs A pointer to a structure to receive the Freeway and ICP protocol-specific parameters required for *Raw* operations. If pOptArgs is NULL, only data from the remote data link application is forwarded to your application. If this field is not NULL, DLI fills the pOptArgs structure with information related to the Freeway server as well as the data link protocol specifics, and forwards all responses from the Freeway server to your application, with the exception of the Freeway open and close responses. If the session is using non-blocking I/O, your application must not reuse this buffer until the dlRead request is complete. See Section 4.1.3.3 on page 83 for information on the optional arguments structure.

Returns

For blocking I/O, a successful dlRead returns the number of bytes read.

For non-blocking I/O, if the AsyncIO and AlwaysQIO parameters are both set to “yes” in the DLI configuration file for a given session, dlRead returns zero (0) if DLI successfully queues the I/O request to its internal I/O queue.

Protocol-specific status and error codes originating at the ICP are returned in two ways: First, the code is returned in a global variable called iICPStatus when the read completes. This global status is available to applications using both blocking and non-blocking I/O. Second, if the application provides the DLI_OPT_ARGS parameter, the code is also returned in the iICPStatus field of the DLI_OPT_ARGS structure.

For any error condition, the dlRead return code is ERROR, and dlerrno contains one of the following error codes (listed alphabetically):

DLI_EWOULDBLOCK The session was configured for non-blocking I/O, and no data could be read immediately.

Action: Use dlPoll to check if your request completed. You might wish to program your application to be awakened by your own IOCH that you provided when you called the dlInit function or this function. Refer to Section 2.2 on page 32 for information on non-blocking I/O.

DLI_READ_ERR_BUF_MUST_BE_NULL Your application has requested a read for a session that is configured to share a TSI connection with the Freeway server.

Action: You must set the buffer for this request to NULL to complete the request.

DLI_READ_ERR_INTERNAL_DLI_ERROR The DLI input queue is corrupted, or an invalid status was encountered (blocking I/O only).

Action: Restart the application and notify Protogate.

DLI_READ_ERR_INVALID_BUF Your application provided a NULL parameter value in place of ppBuf. This parameter must not be NULL; however, it can specify the address of NULL pointer.

Action: Review your application and try again.

DLI_READ_ERR_INVALID_LENGTH Your requested read length (iBufLen) must be greater than or equal to zero (0) and less than or equal to the maximum buffer length allowed by DLI. In *Raw* operation if the pOptArgs parameter is provided, iBufLen can be zero; otherwise, it must be greater than zero.

Action: Call dlPoll using the DLI_POLL_GET_SESS_STATUS option to obtain the maximum buffer size allowed by the DLI (usMaxSessBufSize field of the DLI_SESS_STAT structure, page 82). Review your application and try again.

DLI_READ_ERR_INVALID_SESSID Your session ID is no longer valid.

Action: Review your application and try again.

DLI_READ_ERR_INVALID_STATE This session is not in a proper state to accept a read request.

Action: Review your application and try again.

DLI_READ_ERR_IO_FATAL The DLI encountered a fatal I/O error. The application can assume the connection has been terminated.

Action: Cancel all outstanding read/write requests and free the buffers. Close the session. After the session is closed, another session connection can be attempted. Review the TSI and DLI log files for specific information about the error.

DLI_READ_ERR_NEVER_INIT DLI was never initialized. Your application must initialize DLI using dlInit before using the DLI services.

Action: Review your application.

DLI_READ_ERR_OVERFLOW Your current read request has overflowed the maximum buffer size. The buffer returned contains as much of the data as would fit, and the remainder has been discarded.

Action: Review your TSI configurations for both the client and Freeway. Ensure that the maxBufSize TSI configuration parameters (page 148) are defined as intended. Increase the buffer size supplied to dlRead up to the maximum defined for the session.

DLI_READ_ERR_QADD_FAILED DLI failed to add your request to its internal I/O queues for this session.

Action: Severe error; terminate your application and try again.

DLI_READ_ERR_QFULL DLI cannot accept more read requests, because its input queue is full.

Action: your application must remove complete or timed-out read requests before it can request more reads. Review your application and handle this error accordingly.

DLI_READ_ERR_READ_ERROR A read error occurred other than those currently documented.

Action: Save the DLI and TSI log files and notify Protogate.

DLI_READ_ERR_TIMEOUT Your current read request is timed out.

Action: Consider increasing the timeout TSI configuration parameter (page 149) and try again.

DLI_READ_ERR_TOO_MANY_ERRORS This session has a large number of I/O errors that exceeded the maximum number of errors allowed.

Action: Consider increasing the maximum I/O errors DLI configuration parameter (maxErrors, page 64). Review your operating environment.

DLI_READ_ERR_TSI_BUFF_MISSING TSI failed to return a read buffer to DLI. Severe internal error.

Action: Save the DLI and TSI log files and notify Protogate.

DLI_READ_ERR_UNBIND The connection supporting this session between Freeway and the client application has been closed. The system has performed “Unbind” processing. The connection was closed either because of a “Force Unbind” received from the peer entity (Freeway or client), or because of a failure with the I/O connection.

Action: Cancel all outstanding read/write requests and free the buffers. Close the session. After a successful close, another session connection can be attempted. Examine the message logs on the peer system (if the error occurred in the client application, examine the Freeway log).

For additional error codes, refer to Appendix B.

4.13 dlSyncSelect

The dlSyncSelect function queries a set of session IDs for a read data available condition. This feature is available only for clients in a Freeway server environment (it is not supported in an embedded ICP environment) using blocking I/O. The client application can query a session(s) for read data, and if available, perform the read operation without blocking. This operation does not block; it interrogates the system for read data available and immediately returns this status to the user.

The user builds a session ID array (sessIDArray) containing the list of sessions for which read availability is requested. The number of sessions can be from 1 to the defined maximum number of sessions (see the DLI maxSess parameter on page 63). Session IDs must begin at position 0 in the array (first position in the array), and be packed (no non-used positions). The contents of this array are not modified by the interface. The number of session IDs packed in this array is passed in iNbrSessID. In addition, a result array is passed which will contain the returned read availability status of the sessions in the corresponding array position of the session ID array. A session's availability status is either TRUE (data available) or FALSE (data not available).

Synopsis

```
int dlSyncSelect (
        int     iNbrSessID,         /* # of session ids in sessIDArray        */
        int     sessIDArray[],      /* packed array of session ids for        */
                                    /*    requested read data status          */
        int     readStatArray[]);   /* array containing read data status      */
                                    /*     for sessions in sessIDArray        */
```

Parameters

int iNbrSessID The number of session IDs to be queried in the following session ID array. If a value of 0 is passed (no session IDs to be queried), the function returns zero (0).

int sessIDArray[] An array containing the session IDs whose read availability status is requested. The session IDs are those returned from dlOpen. Session ids must begin at position 0 (the first array element), and be packed (no non-used positions). These values are not modified by the call.

int readStatArray[] An array passed to the interface for the returned TRUE/FALSE read availability status for sessions in the corresponding positions of the session ID array (sessIDArray). This array is modified by the interface. If an error occurs in the call, the contents of this array are indeterminable; all elements should be ignored.

Returns

If the dlSyncSelect function completes successfully, it returns the number of sessions in the session ID array that have read data available (if 3 of 7 sessions in the array have read data available, a value of 3 is returned). If no sessions have data available, 0 is returned.

Successful completion also returns the readStatArray with a TRUE/FALSE value in each position corresponding to the session ID in the session ID array. TRUE means that session has data available; FALSE means data is not currently available. If the function returns a 0 or ERROR, values in this array are indeterminable; they should be ignored. If this function is successful, it modifies iNbrSessID positions in this array.

For any error condition, the dlSyncSelect return code is ERROR, and dlerrno contains one of the following error codes (listed alphabetically):

DLI_SYNCSELECT_ERR_INVALID_ARRAY An input array (sessIDArray or readStatArray) is NULL.

Action: User must supply a valid array.

DLI_SYNCSELECT_ERR_INVALID_SESSID The session ID(s) in the session ID array (sessIDArray) is less than zero, or greater than the maximum allowed sessions defined in the DLI configuration file.

Action: Review the session IDs in the call. These session IDs are those returned from a successful dlOpen request.

DLI_SYNCSELECT_ERR_INVALID_STATE A session(s) in the session ID array (sessIDArray) is not in the proper state to accept this request. Sessions must be "opened" (in the "ready" state) before this operation can be performed. This error is returned to the embedded application if the dlSyncSelect operation is attempted.

Action: Ensure all sessions in the session ID array have successfully opened.

DLI_SYNCSELECT_ERR_NEVER_INIT DLI has not been initialized. The application must perform DLI initialization using dlInit before requesting this service.

Action: Review your application and ensure the previous dlInit was successful.

DLI_SYNCSELECT_ERR_NOT_SYNC A session(s) in the session ID array (sessIDArray) is not defined as "sync". This operation is not valid on sessions defined as "async."

Action: Review your DLI configuration file for correct session definition.

DLI_SYNCSELECT_ERR_TSI_ERROR An error occurred in TSI while attempting this operation.

Action: Review the TSI log file for the specific error, and take corrective action.

Example

One session is open, dlOpen returned with a session ID of 4.

```
sessIDArray[0] = 4;
if ( (nbrReads = dlSyncSelect( 1, sessIDArray, readStatArray ) ) == ERROR )
{
        error processing
}
if ( nbrReads )     /* with only one read in array, we need not look further */
{
        if ( readStatArray[0] == TRUE )
        {
                /* process read available for session sessIDArray[0] – dlRead */
        }
}
```

With multiple sessions in array, go through readStatArray iNbrSessID times or until nbrReads of TRUE are found.

4.14 dlTerm

The dlTerm function closes all sessions and frees all DLI-related system resources. Under normal conditions your application should call dlClose to close all active sessions before calling dlTerm and exiting to the operating system. You should also make an effort to call dlTerm when your application ends abnormally.

The dlTerm function can be called at any time during the life of your application. To use DLI again, you must call dlInit to re-establish the DLI operating environment. It is not recommended that you call this function too often in your application because of the timing cost associated with it. However, in some applications this capability might be essential if your system and network resources are scarce and your application is not time-critical. If you call this function while there are active sessions, DLI issues a forced dlClose on the active sessions before it brings down its service structure. Issuing dlTerm while active sessions exist should be the last option.

Note

The successful writing of client trace files to the client file system requires successful completion of the dlTerm function. When the client application abnormally terminates, DLI trace files are not written.

Synopsis

```
int     dlTerm ( void );
```

Parameters

None

Returns

If this function completes successfully, it returns OK. Otherwise it returns ERROR, and dlerrno contains one of the following error codes (listed alphabetically):

DLI_TERM_ERR_ACT_REM_FAILED DLI failed to terminate its internal active session queue.

Action: Severe error; terminate your application and try again.

DLI_TERM_ERR_ACT_TERM_FAILED DLI failed to terminate its internal active session queue.

Action: Severe error; terminate your application and try again.

DLI_TERM_ERR_CLOSE_FAILED DLI failed to close an active session.

Action: Review the DLI session log, terminate your application and try again.

DLI_TERM_ERR_LOG_END_FAILED DLI failed to terminate its internal logging and tracing facility.

Action: Check your logging and tracing related parameters in the currently used DLI configuration file.

DLI_TERM_ERR_NEVER_INIT DLI was never initialized before.

Action: Review your application and try again.

DLI_TERM_ERR_RES_FREE_FAILED DLI failed to free session-related resources.

Action: Review the DLI session log, terminate your application and try again.

DLI_TERM_ERR_TSI_TERM_FAILED DLI failed to terminate TSI services.

Action: Review your TSI configuration services and TSI error log.

For additional error codes, refer to Appendix B.

4.15 dlWrite

Typically the dlWrite function is used in *Normal* operation to interact with Freeway only for the purpose of sending data to the remote data link application; in this case the optional arguments parameter is set to NULL, and the protocol-specific writeType DLI configuration parameter (page 66) specifies the type of data. However, if you need to request status or configuration information from Freeway or send protocol-specific data, your application must issue a *Raw* dlWrite by providing the optional arguments specifying the protocol specifics.

The following points apply to sending dlWrite requests to Freeway:

- In *Raw* operation, your application must not specify the Freeway server open and close session commands.
- In *Normal* operation, in addition to the Freeway server open and close session commands, your application must not specify any command that would affect the operational status of your data link connection, such as a stop link command.
- Whether your application is configured for *Normal* or *Raw* operation, it can use the DLI_OPT_ARGS structure to specify a *Raw* dlWrite to the Freeway server.
- The protocol-specific localAck DLI configuration parameter (page 64), specifies whether the DLI manages the local data acknowledgment internally for every dlWrite of WAN data.

Note

When using non-blocking I/O, a read request must be queued to receive the local acknowledgment. The read buffer associated with this request remains queued.

Synopsis

```
int dlWrite (
        int             iSessionID,           /* Session ID returned from dlOpen          */
        char            *pBuf,                /* Source buffer for transfer               */
        int             iBufLen,              /* Number of bytes to transfer              */
        int             iWritePriority,       /* Normal or expediting queueing            */
        DLI_OPT_ARGS *pOptArgs);              /* Optional arguments                       */
```

Parameters

int iSessionID This session ID uniquely identifies an active session serviced by DLI. This ID is returned from a dlOpen or dlListen function call.

char *pBuf The address of a buffer whose contents are sent to Freeway or a remote data link application. The buffer can be allocated using the dlBufAlloc function or by a similar C function. However, if the buffer is allocated by a function other than dlBufAlloc, your application must provide sufficient header space for the DLI to store its internal information related to this buffer (usOverhead field on page 79). This field must not be NULL. If the session is using non-blocking I/O, your application must not reuse this buffer until the dlWrite request is complete. See Section 2.4 on page 40 for buffer management information.

int iBufLen The number of bytes to be sent by DLI. This value must not be larger than the maximum buffer size allowed by DLI. After calling dlOpen, call dlPoll using the DLI_POLL_GET_SESS_STATUS option to obtain the maximum application data buffer size allowed by the DLI (usMaxSessBufSize field of the DLI_SESS_STAT structure, page 82). In *Raw* operation, this field can be zero if pOptArgs is used (for example, report requests that do not include data to send).

int iWritePriority The priority of the write operation which applies only to non-blocking I/O (the value does not matter for blocking I/O). Your application can use this field to expedite a request to the Freeway server or to the remote data link application. The default type is a normal write operation. Valid types are:

DLI_WRITE_EXPEDITE If this type is used, your current request is inserted before any output requests whose actual output operation has not started and after any output request that has already started or that was issued with DLI_WRITE_EXPEDITE. This exercises the priority queue concept.

DLI_WRITE_NORMAL If this type is used, your output request is added to the end of the session internal output queue. This exercises the FIFO concept of queue.

DLI_OPT_ARGS *pOptArgs A pointer to a structure that contains the Freeway and ICP protocol-specific parameters required for *Raw* I/O operations. DLI uses the information provided in the pOptArgs structure to fill the header areas of the data buffers. See Section 4.1.3.3 on page 83 for information on the optional arguments structure.

Returns

If successful, the dlWrite function returns the number of bytes written, with one exception. If the AsyncIO and AlwaysQIO parameters are both set to “yes” in the DLI configuration file for a given session, dlWrite returns zero (0) if DLI successfully queues the I/O request to its internal I/O queue. If dlWrite is unsuccessful, the return code is ERROR; and dlerrno contains one of the following error codes (listed alphabetically):

DLI_EWOULDBLOCK The session was configured for non-blocking I/O, and no data could be written immediately.

Action: Use dlPoll to check if your request completed. You might wish to program your application to be awakened by your own IOCH that you provided when you called the dlInit function or this function. Refer to Section 2.2 on page 32 for information on non-blocking I/O.

DLI_WRIT_ERR_BUFA_FAILED The DLI could not allocate a buffer for the local acknowledgment (blocking I/O only).

Action: Check that the application is releasing buffers properly. Consider increasing the maxBuffers TSI configuration parameter (page 148).

DLI_WRIT_ERR_ILLEGAL_ICP_PROT_CMD Your application attempted to send a restricted command to the ICP or the protocol service on the ICP.

Action: Correct your request or use DLI in *Raw* operation to satisfy your request.

DLI_WRIT_ERR_ILLEGAL_SERVER_CMD Your application attempted to send a restricted command to Freeway.

Action: Correct your request or use TSI directly (bypass DLI completely) to satisfy your request.

DLI_WRIT_ERR_INTERNAL_DLI_ERROR The DLI output queue is corrupted or an invalid status was encountered (blocking I/O only).

Action: Restart the application and notify Protogate.

DLI_WRIT_ERR_INVALID_BUF Your application called this function with a NULL pBuf pointer.

Action: Correct your application and try again.

DLI_WRIT_ERR_INVALID_LENGTH The buffer length (iBufLen) must be greater than zero (0) and not greater than the maximum buffer length allowed by DLI. In *Raw* operation if the pOptArgs parameter is provided, iBufLen can be zero; otherwise, it must be greater than zero.

Action: Call dlPoll using the DLI_POLL_GET_SESS_STATUS option to obtain the maximum application data buffer size allowed by the DLI (usMaxSessBufSize field of the DLI_SESS_STAT structure, page 82). Correct your application and try again.

DLI_WRIT_ERR_INVALID_SESSID   Your session ID is no longer valid.

Action: Review your log, terminate your application and try again.

DLI_WRIT_ERR_INVALID_STATE   This session is not in a proper state to accept a write request.

Action: Review your application and try again.

DLI_WRIT_ERR_INVALID_WRITE_TYPE   dlWrite allows either DLI_WRITE_NORMAL or DLI_WRITE_EXP.

Action: Review your application and try again.

DLI_WRIT_ERR_IO_FATAL   The DLI encountered a fatal I/O error. The application can assume the connection has been terminated.

Action: Cancel all outstanding read/write requests and free the buffers. Close the session. After the session is closed, another session connection can be attempted. Review the TSI and DLI log files for specific information about the error.

DLI_WRIT_ERR_LOCAL_ACK_ERROR   A local acknowledgment for a write was not received from the ICP.

Action: Review the operating environment.

DLI_WRIT_ERR_NEVER_INIT   DLI was never initialized. Your application must initialize DLI (dlInit) before it can use it.

Action: Review your application.

DLI_WRIT_ERR_QADD_FAILED   DLI failed to add your write request to its internal I/O queues for this session.

Action: Severe error; terminate your application and try again.

DLI_WRIT_ERR_QFULL DLI cannot accept more write requests because its output queue is full.

Action: Your application must remove complete or timed-out write requests before it can request more writes. Review your application and handle this error accordingly.

DLI_WRIT_ERR_TIMEOUT Your current write request is timed out.

Action: Consider increasing the timeout TSI configuration parameter (page 149) and try again.

DLI_WRIT_ERR_TOO_MANY_ERRORS This session has a large number of I/O errors that exceeded the maximum number of errors allowed.

Action: Consider increasing the maximum I/O errors DLI configuration parameter (maxErrors, page 64). Review your operating environment.

DLI_WRIT_ERR_UNBIND The connection supporting this session between Freeway and the client application has been closed. The system has performed "Unbind" processing. The connection was closed either because of a "Force Unbind" received from the peer entity (Freeway or client), or because of a failure with the I/O connection.

Action: Cancel all outstanding read/write requests and free the buffers. Close the session. After a successful close, another session connection can be attempted. Examine the message logs on the peer system (if the error occurred in the client application, examine the Freeway log).

DLI_WRIT_ERR_WRITE_ERROR The ICP could not send the data to the remote system.

Action: The global variable iICPStatus and the optional arguments iICPStatus field contain the ICP error number.

For additional error codes, refer to Appendix B.

Chapter 5 Tutorial Example Programs

The example programs in this chapter, along with the supporting DLI and TSI text configuration files, will help you get started writing your client application using the DLI functions described in Chapter 4.

The example in Section 5.1 uses blocking I/O, and the example in Section 5.2 uses non-blocking I/O. Both programs use *Normal* operation (described in Section 2.3.1 on page 35) and are based on the Freeway environment shown in Figure 5–1.

The code segment in Section 5.3 on page 174 illustrates *Raw* operation (described in Section 2.3.2 on page 38) to request and receive a protocol-specific report.

The example program in Section 5.4 on page 179 illustrates the dlControl function (described in Section 4.5 on page 95) to reset and download protocol software to the ICP.

The example program in Section 5.5 on page 182 illustrates how to use the DLI usMaxSessBufSize field obtained by calling dlPoll with the DLI_POLL_GET_SESS_STATUS option (described in Section 4.10 on page 114 and Section 4.1.3.2 on page 80).

Note

The preliminary versions of this guide used the term “synchronous” for blocking I/O and “asynchronous” for non-blocking I/O. Some parameter and file names reflect the previous terminology.

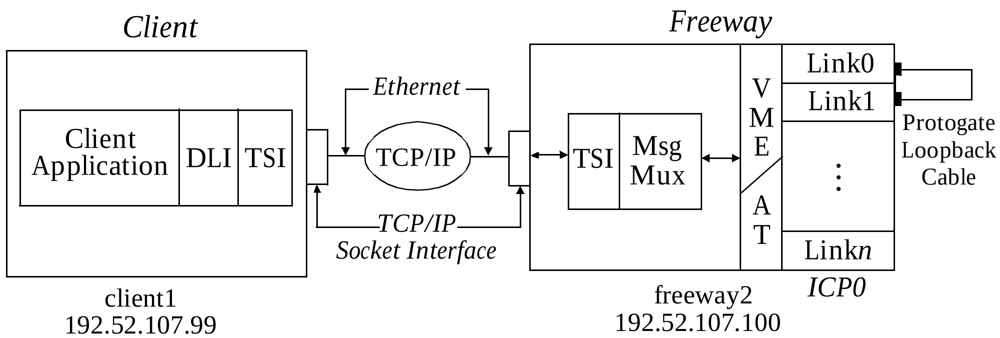

Figure 5–1: Environment for Example Programs

Configuration plays a very important role in both DLI and TSI services, and you should understand the details of Chapter 3 before you begin writing your application. The example programs in Section 5.1 and Section 5.2 demonstrate the following:

1. Initialize DLI services (dlInit).
2. Open the DLI session. This example opens two sessions, one for FMP link 0 and the other for FMP link 1 (dlOpen).
3. Utilize DLI buffer management services (dlBufAlloc and dlBufFree).
4. Read and write data. This example writes your input from the keyboard to link 0, then reads the loopback data from link 1 (dlRead and dlWrite).
5. Close the DLI session (dlClose).
6. Terminate the DLI services (dlTerm).

In addition, the non-blocking I/O example also demonstrates how to:

1. Use your I/O completion handler (IOCH) to receive notification of completed DLI requests.

2. Poll for the status of an outstanding I/O request (dlPoll).

5.1 Example Program using Blocking I/O

For the example program[1] using blocking I/O, there are three code examples provided:

fmpssdcfg	DLI text configuration file input to the dlicfg preprocessor program to create the fmpssdcfg.bin file
fmpsstcfg	TSI text configuration file input to the tsicfg preprocessor program to create the fmpsstcfg.bin file
fmpssp.c	Example application program using blocking I/O

5.1.1 DLI Configuration for Blocking I/O and Normal Operation

The DLI text configuration file defines the sessions your application will use. The fmpssdcfg file shown in Figure 5–2 is used for the blocking I/O example program. You need to specify only those parameters whose values are different from the defaults.

The "main" section starting at the top of Figure 5–2 specifies the DLI configuration for non-session-specific operations. Refer to Table 3–1 on page 63 for an explanation of all parameters.

The "main" section is followed by two session-definition sections for Link00 and Link01 Link00 defines the characteristics of link 0 on ICP 0, and Link01 defines the characteristics of link 1 on ICP 0. Refer to Table 3–2 on page 64 for an explanation of each parameter. If you need to change the default values of any of the protocol-specific ICP link configuration parameters, they should be added to the two session-definition sections at line 19 and line 34 of Figure 5–2.

After your DLI text configuration file is complete, run the dlicfg preprocessor program to create the fmpssdcfg.bin file used by dlInit. Chapter 3 gives an overview of the DLI configuration process. Refer to your particular programmer's guide for the protocol specifics.

1. File name conventions are described under "Document Conventions" in the *Preface*.

Note

The protocol and mode DLI parameters are protocol specific. Refer to your protocol programmer's guide for valid values. This example is written for the FMP data link protocol using the *Shared Manager* ICP access mode. Setting protocol to "FMP" (rather than "raw") causes the session to be opened for *Normal* operation.

```
main                                        // DLI "main" section:          //
{
    TSICfgName = "fmpsstcfg.bin";  // TSI binary configuration file //
    LogLev = 7;
}

//-------------------------------------------------------------------//
// Definition for a FMP Link                                         //
//-------------------------------------------------------------------//
Link00                                      // First session name:          //
{
    Protocol = "FMP";                       // FMP session type        //
    LogLev = 7;
    Transport = "Client";                   // Transport connection name   //
                                            // defined in TSICfgName file  //
    Mode = "shrmgr";                        // Mode of operation for ICP   //
    BoardNo = 0;                            // ICP board number -- based 0 //
    PortNo = 0;                             // link number.            //
}


//-------------------------------------------------------------------//
// Definition for a FMP Link                                         //
//-------------------------------------------------------------------//
Link01                                      // Second session name:         //
{
    Protocol = "FMP";                       // FMP session type        //
    LogLev = 7;
    Transport = "Client";                   // Transport connection name   //
                                            // defined in TSICfgName file  //
    Mode = "shrmgr";                        // Mode of operation for ICP   //
    BoardNo = 0;                            // ICP board number -- based 0 //
    PortNo = 1;                             // link number.            //
}

```

Figure 5–2: DLI Text Configuration File for Blocking I/O (fmpssdcfg)

5.1.2 TSI Configuration for Blocking I/O

The TSI text configuration file defines the transport connections your application will use. The fmpsstcfg file shown in Figure 5–3 is used for the blocking I/O example program. You need to specify only those parameters whose values differ from the defaults.

The "main" section starting at the top of Figure 5–3 specifies the TSI configuration for non-connection-specific operations. The "main" section is followed by one connection-definition section named **Client**. Only one TSI connection definition is needed since the same transport characteristics are used by both DLI sessions. Therefore, **Client** is used as the value of the DLI transport parameter for both sessions previously defined in Figure 5–2 on page 145.

Table 5–1 and Table 5–2 describe the TSI configuration parameters that are mentioned elsewhere in this document or that are used in the blocking I/O example. Refer to the *Freeway Transport Subsystem Interface Reference Guide* for a complete list of TSI configuration parameters.

After your TSI text configuration file is complete, run the tsicfg preprocessor program to create the fmpsstcfg.bin file, which is referenced in the "main" section of the DLI text configuration file (Figure 5–2 on page 145). Chapter 3 gives an overview of the configuration process. Refer to your particular programmer's guide for the protocol specifics.

```
main                                        // TSI "main" section:              //
{
    LogLev = 7;
    maxBuffers = 4096;
    maxBufsize = 1024;
    traceName = "syncTSI.trc";
    traceSize = 64000;
    maxConns = 10;
}

//-----------------------------------------------------------------//
// connection definition.                                          //
//-----------------------------------------------------------------//
Client                                      // First connection name:           //
{
    Transport = "tcp-socket";
    logLev = 7;
    traceLev = 3;
    Server = "freeway2";
    wellKnownPort = 0x2010;
}

```

Figure 5–3: TSI Text Configuration File for Blocking I/O (fmpsstcfg)

Table 5–1: TSI “main” Parameters

TSI Parameter	Default	Valid Values	Description
asyncIO	“no”	boolean	Defines whether TSI uses blocking or non-blocking I/O. The default is “no” (blocking I/O).
logLev	0	0–7	An integer value defining the level of logging the TSI performs and stores in the file name defined by the logName parameter. A higher level specifies more detailed logging; 0 specifies no logging.
logName	“tsilog”	string (ð 80)	A string of characters defining the name (path) of the file for storing the TSI logging information. If the path is not included, the current directory is assumed.
maxBuffers	1024	256–4096	An integer value specifying the maximum number of buffers to be allocated by the TSI during run time for the TSI buffer pool. To prevent your application running out of buffers, take care when you specify maxBuffers to consider the number of TSI connections you need and the queue sizes (MaxInQ and MaxOutQ described in the *Freeway Transport Subsystem Interface Reference Guide*).
maxBufSize	1024	1–64000	An integer value specifying the maximum size of each buffer in the TSI buffer pool. See Section 2.4 on page 40
maxConn	1024	1–1024	Defines the maximum number of connections that TSI has to manage for your application. The example needs only two connections but is configured for 10.
traceLev	0	0–31	An integer value defining the level of tracing (or the sum of several levels) which the TSI performs. See also Appendix D. 0 = no trace 1 = read only 2 = write only 4 = interrupt only 8 = application IOCH 16 = user’s data
traceName	“tsitrace”	string (ð 80)	A string of characters defining the name (path) of the file for storing the TSI tracing information. If the path is not included, the current directory is assumed.
traceSize	0	512–1048576	An integer value specifying the size of the internal trace buffer. The default is zero (tracing is not performed). The smallest allowable size is 512.

Table 5–2: TSI Connection-Related Parameters

TSI Parameter	Default	Valid Values	Description
asyncIO	“no”	boolean	Defines whether TSI uses blocking or non-blocking I/O. The default asyncIO value is “no” (blocking I/O).
logLev	0	0–7	An integer value defining the level of logging the TSI performs for this connection. A higher level specifies more detailed logging, while 0 specifies no logging.
maxBufSize	maxBufSize defined in TSI “main”	1 to maxBufSize defined in “main”	An integer value specifying the maximum data size of the TSI buffers for this connection only. The value must be less than or equal to the “main” entry. The default value is the size specified in the “main” section.
Server	none	string (ð 20)	Defines the name of the TCP/IP server with which the client application communicates. The example program connects to the freeway2 server. The Server name can also be defined in Internet address format. For example, you can define Server = “192.52.107.100”. TSI understands both methods.
timeout	60	0–63999	An integer value specifying the number of seconds the TSI uses to time activities within this connection.
traceLev	0	0–31	An integer value defining the level of tracing (or the sum of several levels) which the TSI performs for this connection. See also Appendix D. 0 = no trace 1 = read only 2 = write only 4 = interrupt only 8 = application IOCH 16 = user’s data
transport	no default (must be specified)	string (ð 20)	A string of characters specifying the transport interface to be used by this connection. There are no defaults. Supported transport interfaces include “tcp-socket” for TCP/IP sockets and “shared-memory” for VxWorks shared memory (server-resident applications).
wellKnownPort	0x2010	5001, 32676	Defines the TCP/IP well-known port where the MsgMux is listening for a connection. The example uses the default value that is the hex value of 2010.

5.1.3 Blocking I/O Example Code Listing

In this section you should refer to Figure 5–4 on page 154 through page 156 as the following steps are explained:

Step 1: Initialize the DLI Services (dlInit)

To begin using DLI, you must first call dlInit to initialize and set up the DLI operating environment. Line 33 on page 154 illustrates dlInit for an application using blocking I/O. The first parameter is the name of the binary configuration file (fmpssdcfg.bin) to be used by DLI to start up its services. The second and third parameters are not used for blocking I/O. The dlInit function returns when it completes because it does not involve any I/O operations.

Step 2: Open a Session with the Remote Application (dlOpen)

To begin communications with the remote application, your application must first call dlOpen. Line 39 and line 47 on page 155 open two sessions with Link00 and Link01 for *Normal* operation. Link00 is configured to be link 0 of ICP 0 of the Freeway server named freeway2. Link01 is configured to be link 1 of ICP 0 of freeway2. If both open requests return successfully, the example begins immediately to send and receive data on the WAN. The second parameter is not used for blocking I/O and therefore is NULL.

Note

Because the cfgLink and enable DLI configuration parameters (page 64) both default to "yes" and were not changed in the DLI configuration file (Figure 5–2 on page 145), the dlOpen requests also configure and enable the ICP links. In this example, the default protocol-specific link options are used since there were no link parameters listed in the DLI configuration file on page 145.

Step 3: Allocate a Buffer for Writing (dlBufAlloc)

Line 56 on page 155 allocates one fixed-size buffer for the example program to use for the dlWrite request. Your application must always provide a buffer for write requests, in contrast to dlRead requests which have the option of letting the DLI provide the read buffer.

Step 4: Send Data using Normal Operation (dlWrite)

After a session has been opened and the link enabled, the client application can exchange data with the remote application. A dlWrite without the optional arguments parameter (*Normal* operation) requests the ICP to send a single data packet to the remote application. The type of data sent depends on the writeType DLI configuration parameter (page 66) which defaults to "normal" for this example. *The valid* writeType *values are protocol specific.*

Line 75 on page 155 uses dlWrite to write your keyboard input (from line 65) to the WAN. The first parameter is the session ID returned by the dlOpen call. The second parameter (which cannot be NULL) is a pointer to the buffer allocated in Step 3 which contains the data to be sent to Link00. The third parameter is the number of bytes (data only) to be written to the ICP link. The fourth parameter (either DLI_WRITE_NORMAL or DLI_WRITE_EXPEDITE) indicates the write priority of the transmission and applies only to non-blocking I/O (the value does not matter for blocking I/O). The last parameter is a pointer to the optional arguments structure which is NULL for *Normal* operation.

The handling of the localAck *DLI configuration parameter (page 64) is protocol specific.* For every block of data transmitted to the WAN in this FMP example, the client application receives an acknowledgment message from the ICP. This example uses the default value of the localAck DLI configuration parameter (page 64), which is "yes," meaning that the DLI manages the local acknowledgment internally for every dlWrite of WAN data. In this mode, the application's write request is blocked until the local acknowledgment is received.

Note

If your application needs to see this local acknowledgment message, you must first set the localAck parameter to "no" in the DLI configuration file. Your application must then make sure that it reads the local acknowledgment message using a *Raw* dlRead request (with the optional arguments parameter).

The dlWrite function returns the number of bytes written (a positive number). If the request fails to complete, the return code is ERROR (–1), and dlerrno provides additional information.

Step 5: Receive Data using Normal Operation (dlRead)

Line 86 on page 156 shows the use of the dlRead function. The first and third parameters are similar to the dlWrite function. The fourth parameter is the optional arguments which is NULL for *Normal* operation. The second parameter is the ***address of the pointer to the buffer*** to be read from the WAN. On line 85 on page 156 notice that, unlike dlWrite, the pointer to the buffer can be NULL if you want DLI to provide its own buffer for a read from the WAN.

The dlRead function returns the number of bytes read (a positive number). If the request fails to complete, the return code is ERROR (–1), and dlerrno provides additional information.

Note

Line 96 on page 156 illustrates the FMP-specific received data block. Packed data messages begin with a two-byte count, a two-byte FMP sequence number, and an error byte for every data block received, followed by the data portion of the message buffer. The two-byte count (iBytes) includes the sequence number, error byte, and the size of the data in bytes; therefore the data portion begins at pInBuf[5].

Step 6: Free Previously Allocated Buffers (dlBufFree)

Line 98 and line 99 on page 156 free the buffer allocated for the dlWrite request using dlBufAlloc, as well as the buffer that the DLI allocated for the dlRead request. Keep in mind that your application does have the responsibility to free any DLI-allocated read buffers.

Step 7: Close a Session (dlClose)

Lines 100 and 101 on page 156 close the two sessions by calling dlClose with a valid session ID.

Step 8: Terminate DLI Services (dlTerm)

Line 102 on page 156 calls dlTerm to terminate the DLI services. Before calling dlTerm, your application should make sure that all sessions are properly closed; otherwise, DLI closes any active sessions with force mode. Your application can call dlInit after dlTerm to re-establish DLI services while it is running. However, you should try to avoid bringing the DLI services up and down since this is time consuming. If your system resources are scarce, however, you might need this option.

```


#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>

#include    “freeway.h”
#include    “dlidefs.h”
#include    “dliusr.h”
#include    “dlierr.h”
#include    “dliicp.h”
#include    “dliprot.h”


typedef struct      _GLOBAL_STRUCT
{
    short           iTimeToRun;
    time_t          tStartTime;
    BOOLEAN         tfNotified;
    BOOLEAN         tfTerminated;
}                   GLOBAL_STRUCT;

GLOBAL_STRUCT       myGlobalStruct;

int
main ()

{
    int             iSessID0, iSessID1;
    int             iOutBufLen, iBytes;
    PCHAR           pInBuf, pOutBuf, s;

    if (dlInit (“fmpssdcfg.bin”, NULL, NULL) == ERROR)
    {
        fprintf (stdout, “ERROR: dlInit failed %d\n”, dlerrno);
        return ERROR;
    }

```

Figure 5–4: FMP Blocking I/O Example (fmpssp.c)

```
if ((iSessID0 = dlOpen ("Link00", NULL)) == ERROR)
{
   fprintf (stdout, "ERROR: dlOpen failed (Link00) %d\n",
            dlerrno);
   dlTerm ();
   return ERROR;
}

if ((iSessID1 = dlOpen ("Link01", NULL)) == ERROR)
{
   fprintf (stdout, "ERROR: dlOpen failed (Link01) %d\n",
            dlerrno);
   dlClose (iSessID0, DLI_CLOSE_NORMAL);
   dlTerm ();
   return ERROR;
}

if ((pOutBuf = dlBufAlloc (1)) == NULL)
{
   fprintf (stdout, "ERROR: No buffers %d\n", dlerrno);
   dlClose (iSessID0, DLI_CLOSE_NORMAL);
   dlClose (iSessID1, DLI_CLOSE_NORMAL);
   dlTerm ();
   return ERROR;
}

fprintf (stdout, "Enter string to be sent: ");
gets (pOutBuf);
for (s = pOutBuf; *s; ++s)
   if (*s == '\n')
   {
      *s = '\0';
      break;
   }

iOutBufLen = strlen (pOutBuf);
if ((iBytes = dlWrite (iSessID0, pOutBuf, iOutBufLen, DLI_WRITE_NORMAL,
                       (PDLI_OPT_ARGS)NULL)) == ERROR)
{
   fprintf (stdout, "ERROR: dlWrite failed %d\n", dlerrno);
   dlClose (iSessID0, DLI_CLOSE_NORMAL);
   dlClose (iSessID1, DLI_CLOSE_NORMAL);
   dlTerm ();
   return ERROR;
}

```

Figure 5–4: FMP Blocking I/O Example (fmpssp.c) *(Cont'd)*

```
pInBuf = NULL;
if ((iBytes = dlRead (iSessID1, &pInBuf, 256, (PDLI_OPT_ARGS)NULL))
        == ERROR)
{
    fprintf (stdout, “ERROR: dlRead failed %d\n”, dlerrno);
    dlClose (iSessID0, DLI_CLOSE_NORMAL);
    dlClose (iSessID1, DLI_CLOSE_NORMAL);
    dlTerm ();
    return ERROR;
}

fprintf (stdout, “%d bytes received: \”%s\”\n”, iBytes-5, &pInBuf[5]);

dlBufFree (pOutBuf);
dlBufFree (pInBuf);
dlClose (iSessID0, DLI_CLOSE_NORMAL);
dlClose (iSessID1, DLI_CLOSE_NORMAL);
dlTerm ();
exit(0);
}
```

Figure 5–4: FMP Blocking I/O Example (fmpssp.c) *(Cont’d)*

5.2 Example Program using Non-Blocking I/O

For the example program[1] using non-blocking I/O, there are three code examples provided:

fmpasdcfg	DLI text configuration file input to the dlicfg preprocessor program to create the fmpasdcfg.bin file
fmpastcfg	TSI text configuration file input to the tsicfg preprocessor program to create the fmpastcfg.bin file
fmpasp.c	Example application program using non-blocking I/O

5.2.1 DLI Configuration for Non-Blocking I/O and Normal Operation

The DLI text configuration file defines the sessions your application will use. The fmpasdcfg file shown in Figure 5–5 is used for the non-blocking I/O example program. You need to specify only those parameters whose values differ from the defaults.

The “main” section starting at the top of Figure 5–5 specifies the DLI configuration for non-session-specific operations. Refer to Table 3–1 on page 63 for an explanation of all parameters.

The “main” section is followed by two session-definition sections for Link00 and Link01 Link00 defines the characteristics of link 0 on ICP 0, and Link01 defines the characteristics of link 1 on ICP 0. Refer to Table 3–2 on page 64 for an explanation of each parameter. If you need to change the default values of any of the protocol-specific ICP link configuration parameters, they should be added to the two session-definition sections at line 18 and line 32 of Figure 5–5.

After your DLI text configuration file is complete, run the dlicfg preprocessor program to create the fmpasdcfg.bin file used by dlInit. Chapter 3 gives an overview of the DLI configuration process. Refer to your particular programmer’s guide for the protocol specifics.

1. File name conventions are described under “Document Conventions” in the *Preface*.

Note

The protocol and mode DLI parameters are protocol specific. Refer to your protocol programmer's guide for valid values. This example is written for the FMP data link protocol using the *Shared Manager* ICP access mode. Setting protocol to "FMP" (rather than "raw") causes the session to be opened for *Normal* operation.

```
main                                   // DLI "main" section:            //
{
    TSICfgName = "fmpastcfg.bin";      // TSI configuration file name //
}

//----------------------------------------------------------------//
// Definition for a FMP Link                                      //
//----------------------------------------------------------------//
Link00                                 // First session name:            //
{
    Protocol = "FMP";                  // FMP session type         //
    Transport = "Client";              // Transport connection name   //
                                       // defined in TSICfgName file  //
    Mode = "shrmgr";                   // Mode of operation for ICP   //
    BoardNo = 0;                       // ICP board number -- based 0 //
    PortNo = 0;                        // link number.          //
     AsyncIO = "Yes";                  // non-blocking I/O.        //
}

//----------------------------------------------------------------//
// Definition for a FMP Link                                      //
//----------------------------------------------------------------//
Link01                                 // Second session name:           //
{
    Protocol = "FMP";                  // FMP session type         //
    Transport = "Client";              // Transport connection name   //
                                       // defined in TSICfgName file  //
    Mode = "shrmgr";                   // Mode of operation for ICP   //
    BoardNo = 0;                       // ICP board number -- based 0 //
    PortNo = 1;                        // link number.          //
     AsyncIO = "Yes";                  // non-blocking I/O.        //
}

```

Figure 5–5: DLI Text Configuration File for Non-Blocking I/O (fmpasdcfg)

5.2.2 TSI Configuration for Non-Blocking I/O

The TSI text configuration file defines the transport connections your application will use. The fmpastcfg file shown in Figure 5–6 is used for the non-blocking I/O example program. You need to specify only those parameters whose values differ from the defaults.

The “main” section starting at the top of Figure 5–6 specifies the TSI configuration for non-connection-specific operations. The “main” section is followed by one connection-definition section named **Client**. Only one TSI connection definition is needed since the same transport characteristics are used by both DLI sessions. Therefore, **Client** is used as the value of the DLI transport parameter for both sessions previously defined in Figure 5–5 on page 158.

Refer back to Table 5–1 on page 148 and Table 5–2 on page 149 which describe the TSI configuration parameters that are mentioned elsewhere in this document or that are used in the non-blocking I/O example. Refer to the *Freeway Transport Subsystem Interface Reference Guide* for a complete list of TSI configuration parameters.

After your TSI text configuration file is complete, run the tsicfg preprocessor program to create the fmpastcfg.bin file used in the “main” section of the DLI configuration file (Figure 5–5 on page 158). Chapter 3 gives an overview of the configuration process. Refer to your particular programmer’s guide for the protocol specifics.

```
main                                              // TSI “main” section:          //
{
    LogLev = 7;
    maxBuffers = 4096;
    maxBufsize = 1024;
    traceName = “asyncTSI.trc”;
    traceSize = 64000;
    maxConns = 10;
     asyncIO = “yes”;                              // non-blocking I/O             //
}

//--------------------------------------------------------------------//
// connection definition.                                             //
//--------------------------------------------------------------------//
Client                                            // First connection name:       //
{
    Transport = “tcp-socket”;
     asyncIO = “yes”;                              // non-blocking I/O             //
    logLev = 7;
    traceLev = 3;
    Server = “freeway2”;
    wellKnownPort = 0x2010;
}

```

Figure 5–6: TSI Text Configuration File for Non-Blocking I/O (fmpastcfg)

5.2.3 Non-Blocking I/O Example Code Listing

In this section you should refer to Figure 5–7 on page 167 through page 173 as each of the following steps is explained:

> **Note**
>
> This example uses two I/O completion handlers. The IOCH defined by dlInit is called for any DLI-related I/O condition for any session managed by the DLI. The IOCH defined by dlOpen handles I/O completions for a specific session.

Step 1: Initialize the DLI Services (dlInit)

To begin using DLI, you must first call dlInit to initialize and set up the DLI operating environment. Even though dlInit is optional, it is good practice to call it before issuing any other DLI request. The dlInit function returns immediately because it does not involve any I/O operations.

Line 111 on page 169 shows how dlInit is used for an application using non-blocking I/O. The first parameter is the name of the DLI binary configuration file (fmpasdcfg.bin) used to start up DLI services. The second parameter is the address of a pointer that your application wishes DLI to return as the first parameter in your I/O completion handler (it is optional and should be NULL if not used). DLI does not manipulate the data area pointed to by this pointer. The third parameter is your general-purpose application I/O completion handler. This IOCH is called by DLI when any DLI-related I/O condition occurs for any session managed by DLI. *When this routine is called, it does not necessarily mean that there is data for your application.* In short, the second parameter of dlInit becomes the first parameter of the IOCH. Your application should do as little as possible inside the IOCH.

Step 2: Open a Session with the Remote Application (dlOpen)

To begin communications with the remote application, your application must first call dlOpen. Lines 117 and 125 on page 169 open two sessions with Link00 and Link01 for *Normal* operation. Link00 is configured to be link 0 of ICP 0 of the Freeway server named freeway2. Link01 is configured to be link 1 of ICP 0 of freeway2.

There are several differences in the dlOpen function for handling non-blocking I/O:

- The asyncIO DLI configuration parameter (lines 17 and 31 on page 158) and the asyncIO TSI configuration parameter (lines 9 and 18 on page 160) must be set to "yes."

- The second dlOpen parameter is optional and is the session-specific I/O completion handler, which is called by the DLI when there is data *for a particular session*. In this example, only one IOCH (fIOComplete) is provided for both sessions (line 60 on page 168). The IOCH has two parameters. The second parameter of dlInit becomes the first parameter of the IOCH. The second parameter of the IOCH is an integer where DLI stores the ID of the session that has either changed state or has data for your application to process. Again, your application should not stay too long inside the IOCH. If the IOCH is given as a parameter in the dlOpen call, that IOCH is invoked when the session is either successfully established or has failed.

- Even though dlOpen returns a valid session ID immediately, it might not complete right away. In the example, after issuing two dlOpen calls for both the Link00 and Link01 sessions, the application stays in a tight loop (line 133 on page 169) and waits for the sessions to be established, as indicated by two flags which are set by the IOCH when it is invoked by DLI (this occurs when the status of the session changes from "not available" to either DLI_STATUS_READY or DLI_STATUS_FAILED). However, your application could perform other tasks while waiting.

- Inside the IOCH, line 71 on page 168 calls dlPoll with the DLI_POLL_GET_SESS_STATUS option to determine the status of the session.

Note

Because the cfgLink and enable DLI configuration parameters (page 64) both default to "yes" and were not changed in the DLI configuration file (Figure 5–2 on page 145), the two dlOpen requests also configure and enable the ICP links. In this example, the default protocol-specific link options are used since there were no link parameters listed in the DLI configuration file on page 158.

Step 3: Allocate a Buffer for Writing (dlBufAlloc)

Line 176 on page 170 allocates one fixed-size buffer for the example program to use for the dlWrite request. Your application must always provide a buffer for write requests, in contrast to dlRead requests which have the option of letting the DLI provide the read buffer.

Step 4: Send Data using Normal Operation (dlWrite)

After a session has been opened and the link enabled, the client application can exchange data with the remote application. A dlWrite without the optional arguments parameter (*Normal* operation) requests the ICP to send a single data packet to the remote application. The type of data sent depends on the writeType DLI configuration parameter (page 66) which defaults to "normal" for this example. *The valid* writeType *values are protocol specific.*

Line 195 on page 171 uses dlWrite to write your keyboard input (from line 186) to the WAN. The first parameter is the session ID returned by the dlOpen call. The second parameter (which cannot be NULL) is a pointer to the buffer allocated in Step 3 which contains the data to be sent to Link00. The third parameter is the number of bytes (data only) to be written to the ICP link. The fourth parameter (either DLI_WRITE_NORMAL or DLI_WRITE_EXPEDITE) indicates the write priority of the transmission and applies only to non-blocking I/O. The last parameter is a pointer to the optional arguments structure which is NULL for *Normal* operation.

The dlWrite function returns the number of bytes written (a positive number). If the request fails to complete, the return code is ERROR (–1), and dlerrno provides additional information. The dlWrite function might return immediately when data is available, even though it is using non-blocking I/O. Your application must always check the return code and dlerrno to determine if the request is complete or being queued.

If dlWrite returns ERROR, and dlerrno is set to DLI_EWOULDBLOCK (see line 198 on page 171), it means that your request is being queued internally to DLI and will be completed at a later time. When the request completes, DLI calls your IOCH to notify of the I/O completion, and your application can call dlPoll (with the DLI_POLL_WRITE_COMPLETE option as shown on line 219 on page 171) to retrieve the completion status. Your application must not reuse buffers that it gave to the dlWrite function until the request completes.

You should consider the following DLI configuration parameters which affect the operation of dlWrite:

- This example uses the default value of the localAck DLI configuration parameter (page 64), which is "yes," meaning that the DLI internally manages the protocol-specific local data acknowledgment for every dlWrite of WAN data.

- If you prefer that the DLI always queues an I/O request whether or not the request can be satisfied immediately, set the alwaysQIO DLI configuration parameter (page 64) to "yes." This example uses the default of "no," but setting alwaysQIO to "yes" could ease your application implementation.

Step 5: Receive Data using Normal Operation (dlRead)

Lines 163 and 166 on page 170 show the use of the dlRead function. The first and third parameters are similar to the dlWrite function. The fourth parameter is the pointer to the optional arguments structure, which is NULL for *Normal* operation. The second parameter is the ***address of the pointer to the buffer*** to be read from the WAN. On line 165 on page 170 notice that, unlike dlWrite, the pointer to the dlRead buffer can be NULL if you want DLI to use its own buffer for a read from the WAN.

The dlRead function returns the number of bytes read (a positive number). If the request fails to complete, the return code is ERROR (–1), and dlerrno provides additional information. The dlRead function might return immediately when data is available, even though it is using non-blocking I/O. Your application must always check the return code and dlerrno to determine if the request is complete or being queued. If dlRead returns ERROR, and dlerrno is set to DLI_EWOULDBLOCK, it means that your request is being queued internally to DLI and will be completed at a later time.

Note

The example application does not check dlerrno after the dlRead call (line 163 on page 170) because it assumes a loopback condition and queues a read before issuing a write request. Therefore, there will be no data until dlWrite is called.

When the read request completes, DLI invokes your IOCH to notify of the I/O completion, and your IOCH can then call dlPoll (with the DLI_POLL_GET_SESS_STATUS option as shown on line 71 on page 168) to determine the number of I/O completions. Then your application can call dlPoll again (with the DLI_POLL_READ_COMPLETE option as shown on line 224 on page 171) to retrieve the data.

Step 6: Free Previously Allocated Buffers (dlBufFree)

Lines 245 and 246 on page 172 free the buffer allocated for the dlWrite request using dlBufAlloc, as well as the buffer that the DLI allocated for the dlRead request. Keep in mind that your application does have the responsibility to free any DLI-allocated read buffers.

Step 7: Close a Session (dlClose)

Lines 272 and 272 on page 173 close the two sessions by calling dlClose with a valid session ID.

Step 8: Terminate DLI Services (dlTerm)

Line 299 on page 173 calls dlTerm to terminate the DLI services. Before calling dlTerm, your application should make sure that all sessions are properly closed; otherwise, DLI closes any active sessions with force mode. Your application can call dlInit after dlTerm to re-establish DLI services while it is running. However, you should try to avoid bringing the DLI services up and down since this is time consuming. If your system resources are scarce, however, you might need this option.

```

#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>
#include       <time.h>

#include    "freeway.h"
#include    "dlidefs.h"
#include    "dliusr.h"
#include    "dlierr.h"
#include    "dliicp.h"
#include    "dliprot.h"
#include    "gentest.h"


typedef struct         _GLOBAL_STRUCT
{
    short              iTimeToRun;
    BOOLEAN            tfNotified;
    BOOLEAN            tfTerminated;
    int                iSess0Status;
    int                iSess1Status;
    BOOLEAN            tfSess0Data;
    BOOLEAN            tfSess1Data;
    time_t             tStartTime;
    int                iSessID0;
    int                iSessID1;
}                      GLOBAL_STRUCT;
typedef GLOBAL_STRUCT   *PGLOBAL_STRUCT;
GLOBAL_STRUCT           myGlobalStruct;

int iProcID;

/*-------------------------------------------------------------*/
/* This function is invoked by dli when there is any incoming    */
/* data for any session that dli maintains.  When this function  */
/* is invoked, it does not necessarily mean that there is any    */
/* data for the application!                                     */
/*-------------------------------------------------------------*/
int
fNotify ( pUserCB )
    char *         pUserCB;
{
    PGLOBAL_STRUCT  pMyGlobal;

    pMyGlobal = (PGLOBAL_STRUCT) pUserCB;
    fprintf (stdout, "Being notified of I/O event\n");
    pMyGlobal-> tfNotified = TRUE;
    genNotifyMain (iProcID);
    return OK;
}
```

Figure 5–7: FMP Non-Blocking I/O Example (fmpasp.c)

```

/*--------------------------------------------------------------*/
/* This function is invoked by dli when there is any incoming    */
/* data for the given session id (iSessID).  When this function  */
/* is invoked, it means that there is data for the application,  */
/* and for this particular session id.                           */
/*--------------------------------------------------------------*/
int
fIOComplete ( pUserCB, iSessID )
    char *          pUserCB;
    int             iSessID;
{

    DLI_SESS_STAT   sessStat;
    PGLOBAL_STRUCT  pMyGlobal;

    fprintf (stderr, "Notified by sess %d \n", iSessID);
    pMyGlobal = (PGLOBAL_STRUCT) pUserCB;

    dlPoll (iSessID, DLI_POLL_GET_SESS_STATUS, (PCHAR*)NULL, (PINT)NULL,
            (PCHAR)&sessStat, (PDLI_OPT_ARGS*)NULL);
    if (iSessID == pMyGlobal-> iSessID0)
    {
       pMyGlobal-> tfSess0Data = TRUE;
       pMyGlobal-> iSess0Status = sessStat. iSessStatus;
    }
    else
    {
       pMyGlobal-> tfSess1Data = TRUE;
       pMyGlobal-> iSess1Status = sessStat. iSessStatus;
    }

    return OK;
}


```

Figure 5–7: FMP Non-Blocking I/O Example (fmpasp.c) *(Cont'd)*

```
/*--------------------------------------------------------------*/
/* main program:                                                */
/*--------------------------------------------------------------*/
int
main ()

{
    int            iOutBufLen, iBytes;
    time_t         tStart;
    PCHAR          pInBuf, pOutBuf, s;
    DLI_SESS_STAT  sessStat;

    iProcID = getpid();

    signal (SIGALRM, SIG_IGN);
    signal (SIGINT, SIG_IGN);

    myGlobalStruct. tfNotified = FALSE;
    myGlobalStruct. iSess0Status = DLI_STATUS_CLOSED;
    myGlobalStruct. iSess1Status = DLI_STATUS_CLOSED;
    myGlobalStruct. tfSess0Data = FALSE;
    myGlobalStruct. tfSess1Data = FALSE;

    if (dlInit ("fmpasdcfg.bin", (PCHAR) &myGlobalStruct, fNotify) == ERROR)
    {
       fprintf (stdout, "ERROR: dlInit failed %d\n", dlerrno);
       return ERROR;
    }

    if ((myGlobalStruct. iSessID0 = dlOpen ("Link00", fIOComplete)) == ERROR)
    {
       fprintf (stdout, "ERROR: dlOpen failed (Link00) %d\n",
              dlerrno);
       dlTerm ();
       return ERROR;
    }

    if ((myGlobalStruct. iSessID1 = dlOpen ("Link01", fIOComplete)) == ERROR)
    {
       fprintf (stdout, "ERROR: dlOpen failed (Link01) %d\n",
              dlerrno);
       dlClose (myGlobalStruct. iSessID1, DLI_CLOSE_NORMAL);
       dlTerm ();
       return ERROR;
    }
    for ( tStart = time(NULL); sDiffTime (time(NULL),tStart) < 5 &&
        ( myGlobalStruct. iSess0Status != DLI_STATUS_READY ||
        myGlobalStruct. iSess1Status != DLI_STATUS_READY  ) ; )
    {
       SLEEP (1);     /* could do something else here */
```

Figure 5–7: FMP Non-Blocking I/O Example (fmpasp.c) *(Cont'd)*

```
/*----------------------------------------------*/
/* check the status of each session here, since */
/* the fIOComplete routine may have been called */
/* before the myGlobalStruct.iSessID was set,   */
/* and was therefore unable to set the session  */
/* status.                                      */
/*----------------------------------------------*/
    dlPoll (myGlobalStruct. iSessID0, DLI_POLL_GET_SESS_STATUS,
            (PCHAR*)NULL, (PINT)NULL,
            (PCHAR)&sessStat, (PDLI_OPT_ARGS*)NULL);
    myGlobalStruct. iSess0Status = sessStat. iSessStatus;
    dlPoll (myGlobalStruct. iSessID1, DLI_POLL_GET_SESS_STATUS,
            (PCHAR*)NULL, (PINT)NULL,
            (PCHAR)&sessStat, (PDLI_OPT_ARGS*)NULL);
    myGlobalStruct. iSess1Status = sessStat. iSessStatus;

}

myGlobalStruct. tfSess0Data = FALSE;
myGlobalStruct. tfSess1Data = FALSE;

/*-----------------------------------------------------------*/
/* must always keep at least one read posted on each link!   */
/*-----------------------------------------------------------*/
pInBuf = NULL;
iBytes = dlRead (myGlobalStruct. iSessID0, &pInBuf, 256,
                 (PDLI_OPT_ARGS)NULL);
pInBuf = NULL;
iBytes = dlRead (myGlobalStruct. iSessID0, &pInBuf, 256,
                 (PDLI_OPT_ARGS)NULL);

pInBuf = NULL;
iBytes = dlRead (myGlobalStruct. iSessID1, &pInBuf, 256,
                 (PDLI_OPT_ARGS)NULL);
pInBuf = NULL;
iBytes = dlRead (myGlobalStruct. iSessID1, &pInBuf, 256,
                 (PDLI_OPT_ARGS)NULL);

if ((pOutBuf = dlBufAlloc (1)) == NULL)
{
    fprintf (stdout, "ERROR: No buffers %d\n", dlerrno);
    dlClose (myGlobalStruct. iSessID0, DLI_CLOSE_NORMAL);
    dlClose (myGlobalStruct. iSessID1, DLI_CLOSE_NORMAL);
    dlTerm ();
    return ERROR;
}

```

Figure 5–7: FMP Non-Blocking I/O Example (fmpasp.c) *(Cont'd)*

```
fprintf (stdout, "Enter string to be sent: ");
gets (pOutBuf);
for (s = pOutBuf; *s; ++s)
   if (*s == '\n')
   {
      *s = '\0';
      break;
   }

iOutBufLen = strlen (pOutBuf);
if ((iBytes = dlWrite (myGlobalStruct. iSessID0, pOutBuf, iOutBufLen,
                  DLI_WRITE_NORMAL, (PDLI_OPT_ARGS)NULL)) == ERROR)
{
   if (dlerrno != DLI_EWOULDBLOCK)
   {
      fprintf (stdout, "ERROR: dlWrite failed %d\n", dlerrno);
      dlClose (myGlobalStruct. iSessID0, DLI_CLOSE_NORMAL);
      dlClose (myGlobalStruct. iSessID1, DLI_CLOSE_NORMAL);
      dlTerm ();
      return ERROR;
   }

   /*----------------------------------------------------------*/
   /* wait for the write to complete.                          */
   /*----------------------------------------------------------*/
   for ( tStart = time (NULL);
        sDiffTime (time(NULL), tStart) < 5 &&
        !myGlobalStruct. tfSess0Data;
      )
   {
      SLEEP (1);
   }
               /* remove write request from the output queue   */
   pOutBuf = NULL;
   dlPoll (myGlobalStruct. iSessID0, DLI_POLL_WRITE_COMPLETE,
         &pOutBuf, &iBytes, (PCHAR)NULL, (PDLI_OPT_ARGS*)NULL);

   /* remove the write acknowledge packet from link 0's input queue*/
   pInBuf = NULL;
   dlPoll (myGlobalStruct. iSessID0, DLI_POLL_READ_COMPLETE,
         &pInBuf, &iBytes, (PCHAR)NULL, (PDLI_OPT_ARGS*)NULL);
}
```

Figure 5–7: FMP Non-Blocking I/O Example (fmpasp.c) *(Cont'd)*

```
/*----------------------------------------------------------*/
/* wait for the read to complete on link 1.                 */
/*----------------------------------------------------------*/
for ( tStart = time(NULL);
    sDiffTime (time(NULL),tStart) < 5 && !myGlobalStruct. tfSess1Data;
   )
{
   SLEEP (1);
}

pInBuf = NULL;
if ( dlPoll (myGlobalStruct. iSessID1, DLI_POLL_READ_COMPLETE,
       &pInBuf, &iBytes, (PCHAR)NULL, (PDLI_OPT_ARGS*)NULL) == ERROR)
   fprintf (stdout, "dlPoll failed - dlerrno = %d\n", dlerrno);
else
   fprintf (stdout, "%d bytes received: \"%s\"\n", iBytes-5, &pInBuf[5]);

dlBufFree (pOutBuf);
dlBufFree (pInBuf);

/*----------------------------------------------------------*/
/* cancel all reads that may be queued to DLI before we close*/
/* sessions.                                  */
/*----------------------------------------------------------*/
for (pInBuf = NULL;
    dlPoll (myGlobalStruct. iSessID0, DLI_POLL_READ_CANCEL, &pInBuf,
          &iBytes, (PCHAR)NULL, (PDLI_OPT_ARGS*)NULL) == OK;
    pInBuf = NULL
   )
{
   if (pInBuf)
      dlBufFree (pInBuf);
}
for (pInBuf = NULL;
    dlPoll (myGlobalStruct. iSessID1, DLI_POLL_READ_CANCEL, &pInBuf,
          &iBytes, (PCHAR)NULL, (PDLI_OPT_ARGS*)NULL) == OK;
    pInBuf = NULL
   )
{
   if (pInBuf)
      dlBufFree (pInBuf);
}
```

Figure 5–7: FMP Non-Blocking I/O Example (fmpasp.c) *(Cont'd)*

```
fprintf (stderr, "Closing circuits \n");
dlClose (myGlobalStruct. iSessID0, DLI_CLOSE_FORCE);
dlClose (myGlobalStruct. iSessID1, DLI_CLOSE_FORCE);
fprintf (stderr, "Done dlClose\n");

/*---------------------------------------------------------*/
/* wait for both sessions to become closed.                */
/*---------------------------------------------------------*/
tStart = time(NULL);
do
{
   SLEEP(1);
   dlPoll (myGlobalStruct. iSessID0, DLI_POLL_GET_SESS_STATUS,
      (PCHAR*)NULL, (PINT)NULL,
      (PCHAR)&sessStat, (PDLI_OPT_ARGS*)NULL);
} while ( sDiffTime (time(NULL),tStart) < 5 &&
        sessStat.iSessStatus != DLI_STATUS_CLOSED);

tStart = time(NULL);
do
{
   SLEEP(1);
   dlPoll (myGlobalStruct. iSessID1, DLI_POLL_GET_SESS_STATUS,
      (PCHAR*)NULL, (PINT)NULL,
      (PCHAR)&sessStat, (PDLI_OPT_ARGS*)NULL);
} while ( sDiffTime (time(NULL),tStart) < 5 &&
        sessStat.iSessStatus != DLI_STATUS_CLOSED);

dlTerm ();
exit(OK);
}
```

Figure 5–7: FMP Non-Blocking I/O Example (fmpasp.c) *(Cont'd)*

5.3 Using Raw Operation

If your application requires protocol-specific information such as ICP link statistics or link configuration, or performs data transfer requests other than for single packets, it can use *Raw* operation by including the optional arguments parameter in both the dlRead and dlWrite calls. Use of *Raw* operation is recommended whenever you need to interface with the protocol software for any reason outside of simple data transfer.

If your protocol supports *Normal* operation, it is possible to configure and start the protocol in *Normal* operation by specifying the protocol name in the "protocol" field of the DLI configuration file, then use *Raw* operation for all subsequent reads and writes as described above. This method has the advantage of letting DLI handle all the set-up of the protocol while still giving your application the greater control offered by *Raw* operation. If you specify *Raw* in the "protocol" field, your application must handle all the protocol set-up commands such as ATTACH, BIND, and Link Configuration.

Note

Any time a dlRead or a dlWrite request includes the optional arguments parameter, it is considered to be a *Raw* operation.

5.3.1 Optional Arguments Structure

The optional arguments are described in Section 4.1.3.3 on page 83. The fields of the DLI_OPT_ARGS structure shown in Table 5–3 are required for all *Raw* dlWrite requests. The remaining fields should be filled in according to the command's instructions given in your particular protocol programmer's guide.

The Figure 5–8 example code segment uses *Raw* operation to get a link statistics report from the ICP. In this example dlWrite (line 56 on page 177) requests an FMP link statistics report from the FMP protocol service on the ICP by defining an appropriate structure for the report (line 21 on page 176) and specifying the appropriate protocol-specific command in the optional arguments structure (line 49 on page 177). Note that unlike the two earlier examples, this code segment is not complete and requires additional code to run. The use of the optional arguments varies from one request to another and between different protocols (refer to your particular protocol programmer's guide).

Table 5–3: Optional Arguments Required for Raw dlWrite Requests

Optional Arguments Field	Description
usFWPacketType	The type of packet (control or data) being sent to the Freeway server. This is almost always set to FW_DATA.
usFWCommand	The command issued to the msgmux task in the Freeway server. This is almost always FW_ICP_WRITE.
usICPCommand	The command issued to the ICP. When issuing a protocol-specific command, this is usually set to DLI_ICP_CMD_WRITE.
usProtCommand	The protocol-specific command issued to the protocol software. Refer to your particular protocol programmer's guide for a description of the commands available.

```
#include    <stdio.h>
#include    <stdlib.h>
#include    <string.h>

#include    "freeway.h"
#include    "queue.h"
#include    "utils.h"
#include    "dlidefs.h"
#include    "dliusr.h"
#include    "dlierr.h"
#include    "dliicp.h"
#include    "dlifmp.h"
#include    "dliprot.h"


static char     cMsg[1024];
static int      iMsgLen;

#define     FMP_OVERHEAD    5

typedef struct      _PROT_STATISTICS_REPORT
{
    unsigned short  usBCCErrors;            /* block check error */
    unsigned short  usParityErrors;
    unsigned short  usRcvOverrun;
    unsigned short  usQLimitErrors;
    unsigned short  usSent;
    unsigned short  usReceived;
    unsigned short  usBufferNA;
    unsigned short  usBufferOverrun;
}                   FMP_STATISTICS_REPORT;
typedef FMP_STATISTICS_REPORT   *PFMP_STATISTICS_REPORT;
#define FMP_STATISTICS_REPORT_SIZE  sizeof(FMP_STATISTICS_REPORT)

int
fGetStatistics (pSess)
    PSESSION        pSess;
{
    DLI_OPT_ARGS    optArgs;
    PDLI_OPT_ARGS   pOptArgs;
    PCHAR           pBuf;
    int             iBufLen, i;
    PFMP_STATISTICS_REPORT  pStat;

```

Figure 5–8: Link Statistics Report using *Raw* Operation

```
memset ((PCHAR) &optArgs, 0, DLI_OPT_ARGS_SIZE);
optArgs. usFWPacketType = FW_DATA;
optArgs. usFWCommand = FW_ICP_WRITE;
optArgs. usICPCommand = DLI_ICP_CMD_WRITE;
optArgs. usProtCommand = DLI_PROT_GET_STATISTICS_REP;
if ((pBuf = dlBufAlloc (1)) == NULL)
{
   fprintf (stdout, "GetS: no bufs \n");
   return ERROR;
}

if (dlWrite (pSess-> iSessID, pBuf, 0, DLI_WRITE_NORMAL,
          &optArgs) == ERROR)
{
   for (i = 0, pBuf = NULL; i < 2; i++, pBuf = NULL)
      if (dlPoll (pSess-> iSessID, DLI_POLL_WRITE_COMPLETE, &pBuf,
               &iBufLen, (PCHAR)NULL, (PDLI_OPT_ARGS*)NULL)
               == ERROR)
         sleep (1);
      else
         break;
}
if (pBuf)
   dlBufFree (pBuf);

pBuf = NULL;
memset ((PCHAR)&optArgs, 0, DLI_OPT_ARGS_SIZE);

if (dlRead (pSess-> iSessID, &pBuf, pCB-> iMaxBufSize,
          &optArgs) == ERROR)
{
   for (i = 0, pBuf = NULL; i < 2; i++, pBuf = NULL)
      if (dlPoll (pSess-> iSessID, DLI_POLL_READ_COMPLETE, &pBuf,
               &iBufLen, (PCHAR)NULL, &pOptArgs) == ERROR)
         sleep (1);
      else
         break;
}

```

Figure 5–8: Link Statistics Report using *Raw* Operation *(Cont'd)*

```
  if (pBuf && pOptArgs)
  {
    pStat = (PFMP_STATISTICS_REPORT) pBuf;
    fprintf (stdout, "\n%s:\n", pSess-> cSessName);
    fprintf (stdout, "BCC:%5d ParErr %5d RcvOverrun:%5d QLimit:%5d\n",
         pStat-> usBCCErrors, pStat-> usParityErrors,
         pStat-> usRcvOverrun, pStat-> usQLimitErrors);
    fprintf (stdout, "Xmitted:%5d Rcvd:%5d BufNA:%5d BufOverrun:%5d\n",
         pStat-> usSent, pStat-> usReceived,
         pStat-> usBufferNA, pStat-> usBufferOverrun);
    return OK;
  }
  return ERROR;
}

```

Figure 5–8: Link Statistics Report using *Raw* Operation *(Cont'd)*

5.4 Example Program using dlControl

The program shown in Figure 5–9 initializes the DLI/TSI and sends an ICP reset and download request to Freeway. It terminates when the download completes. See Section 4.5 on page 95 for more information about the dlControl function.

```
/**************************************************************************/
/* Test program to initiate a download of an ICP                          */
/*------------------------------------------------------------------------*/
/* This program assumes there is a DLI config file called 'resetdcfg.bin'  */
/* with a session entry called 'RawSess0'                                   */
/**************************************************************************/

#include <stdio.h>

/*----------------------*/
/* DL API include files */
/*----------------------*/
#include "freeway.h"
#include "control.h"
#include "dlidefs.h"
#include "dliusr.h"
#include "dlierr.h"

int cid;
DLBOOLEAN done = FALSE;

int queue_intr();
int notify_intr();
int sighdlr();

/*****************************************************/
/* download main function                            */
/*****************************************************/
main()
{
  int retval;

  /*------------------------*/
  /* initialize the DLI/TSI */
  /*------------------------*/
  retval = dlInit("resetdcfg.bin", (char *)NULL, notify_intr);
  printf("Back from dlInit with retval(%d), dlerrno(%d)\n",retval,dlerrno);
```

Figure 5–9: Example dlControl Program

```
  /*---------------------------*/
  /* send the download command */
  /*---------------------------*/
  retval = dlControl("RawSess0",DLI_CTRL_RESET_ICP,queue_intr);
  printf("Back from dlControl with retval(%d), dlerrno(%d)\n",
  retval,dlerrno);

  /*-----------------------------------------------------*/
  /* wait for download completion. The download request is */
  /* sent from the signal handler                         */
  /*-----------------------------------------------------*/
  while (!done);

  dlTerm();

  printf("Download completed\n");
}

/****************************************************/
/* handler for completed I/O                        */
/****************************************************/
int sighdlr()
{
  DLI_SESS_STAT stat;
  int retval;

  /*------------------------------------*/
  /* get and display the session status */
  /*------------------------------------*/
  printf("Enter sig handler\n");
  retval = dlPoll(cid,DLI_POLL_GET_SESS_STATUS,NULL,0,(char *)&stat,NULL);
  printf("Back from dlPoll with retval(%d), dlerrno(%d)\n",retval,dlerrno);
  printf(" iSessStatus(%d)\n",stat.iSessStatus);
  printf(" iICPMode (%d)\n",stat.iICPMode);
  printf(" iBoardNo (%d)\n",stat.iBoardNo);
  printf(" iPortNo (%d)\n",stat.iPortNo);
  printf(" ver (%s)\n",stat.cServerVer);

  /*-----------------------------------*/
  /* see if the download has completed */
  /*-----------------------------------*/
   if (stat.iSessStatus == DLI_STATUS_READY)
     done = TRUE;
}
```

Figure 5–9: Example dlControl Program *(Cont'd)*

```
/*****************************************************/
/* call back routine for individual I/O completions   */
/*****************************************************/
int queue_intr(intr_data, dli_cid)
char *intr_data;
int dli_cid;
{
  /*-----------------------------*/
  /* save the I/O completion cid */
  /*-----------------------------*/
  cid = dli_cid;
  printf("Enter queue_intr for cid(%d)\n",cid);
}

/*******************************************************/
/* call back routine indicating that all I/O completed */
/*******************************************************/
int notify_intr(intr_data)
char *intr_data;
{
  static int busy = 0;

  printf("enter notify_intr\n");

  /*-----------------------------------*/
  /* protect our selves from re-entrency */
  /*-----------------------------------*/
  if (!busy)
  {
     busy++;
     sighdlr();
     busy--;
  }
}
```

Figure 5–9: Example dlControl Program *(Cont'd)*

5.5 Example dlPoll Using usMaxSessBufSize Field

Figure 5–10 is an example of how to use the DLI usMaxSessBufSize field obtained by calling dlPoll with the DLI_POLL_GET_SESS_STATUS option. A similar approach would apply to use the TSI usMaxConnBufSize field obtained by calling tPoll with the TSI_POLL_GET_CONN_STATUS option.

```
============================================================
#include <stdio.h>
#include <stdlib.h>

/*----------------------*/
/* DL API include files */
/*----------------------*/
#include <freeway.h>
#include <control.h>
#include <dlidefs.h>
#include <dliusr.h>
#include <dlierr.h>
#include <dliprot.h>
#include <dliicp.h>
#include <dlicperr.h>


#define BIN_FILE "appdcfg.bin"

/****************************************************/
/*  main function                                   */
/****************************************************/
main()
{
DLI_SESS_STAT Stat;
int            sessid,
               retval = 0;
char           *pBuf;

   /*---------------------------*/
   /* initialize the DLI/TSI    */
   /*---------------------------*/
   if ( dlInit(BIN_FILE, (char *)NULL, NULL) != OK)
        exit (-1);
```

Figure 5–10: Example dlPoll Program Using usMaxSessBufSize Field

```
  /*--------------------------*/
 /* Open Session "icp0"        */
 /*--------------------------*/
 sessid = dlOpen("icp0", NULL);
 if (sessid == ERROR)
      exit (-1);


 /*-----------------------------------------------------------------*/
 /*  Get session status - now contains session buffer size               */
 /*-----------------------------------------------------------------*/
 dlPoll(sessid, DLI_POLL_GET_SESS_STATUS, (PCHAR*)NULL,
          (int*)NULL,(PCHAR)&Stat, (PDLI_OPT_ARGS*)NULL);

 /*-----------------------------------------------------------------------*/
 /*  Allocate memory for buffer with size  usMaxSessBufSize                        */
 /*-----------------------------------------------------------------------*/
 pBuf = dlBufAlloc(Stat.usMaxSessBufSize);
 ..........
 ..........

dlWrite (icp0, pBuf, Stat.usMaxSessBufSize , DLI_WRITE_NORMAL,
             (PDLI_OPT_ARGS)NULL);
 .........
 .........

 /* ------------------------------------------*/
 /*  We are done, so close the session              */
 /* ------------------------------------------*/
 dlClose( sessid, DLI_CLOSE_FORCE);

 dlTerm();
}
```

Figure 5–10: Example dlPoll Program Using usMaxSessBufSize Field *(Cont'd)*

Appendix

A DLI Header Files

Table A–1 describes the header files you need to develop your DLI application. These files are located in your user installation directory in the freeway/include subdirectory.

Table A–1: DLI Header Files

Header File Name	Description
dlicp.h	ICP command definitions (for *Raw* dlWrite requests)
dlicperr.h	ICP error code definitions (returned in the iICPStatus field of the DLI optional arguments)
dlidefs.h	DLI definitions
dlierr.h	Error code definitions
dli*ppp*.h	Protocol-specific command definitions (for *Raw* dlWrite requests), where *ppp* is the protocol designation (for example, dlifmp.h)
dliprot.h	Generic protocol command definitions (for *Raw* dlWrite requests)
dliusr.h	DLI structures and prototypes
freeway.h	Freeway command definitions

Appendix

B DLI Error Codes

The DLI error codes are defined in the dlierr.h include file. This chapter describes the following:

- Internal error codes (Section B.1)
- Command-specific error codes (Table B–1 on page 195)
- Error handling for dead socket detection (Section B.3 on page 202)

Note

While developing your DLI application, if a particular error occurs consistently, contact Protogate for further assistance.

B.1 Internal Error Codes

The DLI uses the global variable dlerrno to store all its error codes; it offers similar services to errno provided in the C language. Your application should check this value on all returns from DLI function calls. To assist you in debugging your application, the following codes (listed alphabetically) describe internal error conditions of DLI services that are returned in the global variable dlerrno.

DLI_EVTG_ERR_ICP_STAT_ERR   DLI received an invalid status value from the ICP.

Action: Review your trace file and try again.

DLI_CALLBACK_Q_OVRFLOW The DLI queue where callback requests are placed has overflowed. When this occurs, DLI might have failed to deliver a callback.

Action: Revise your application so fewer DLI reads or writes (dlRead/dlWrite) are made from the context of your IOCH; or increase the DLI "main" parameter callbackQsize (page 63) and rebuild the DLI/TSI library.

DLI_EVTG_ERR_IOMUX_FAILED DLI failed to multiplex its I/O requests.

Action: Review the DLI error log for additional error messages.

DLI_IO_ERR_INVALID_STATE DLI encountered an invalid state in its internal state processing machine.

Action: Review the DLI trace and error logs.

DLI_IO_ERR_TOO_MANY_ERRORS DLI encountered too many I/O error conditions.

Action: Review your operating environment and DLI configuration services.

DLI_IOM_ERR_QIN_POLL_ERROR DLI received an error condition when it invoked tPoll on the TSI internal input queue.

Action: Check TSI services and its configuration.

DLI_IOM_ERR_QIN_UPDATE_ERROR DLI failed to update the DLI input buffer for the related session.

Action: Severe error. Check DLI configuration services, terminate your application and try again.

DLI_IOM_ERR_QOUT_POLL_ERROR DLI received an error condition when it invoked tPoll on the TSI internal output queue.

Action: Check TSI services and its configuration.

DLI_IOM_ERR_QOUT_UPDATE_ERROR DLI failed to update the DLI output buffer for the related session.

Action: Severe error. Check DLI configuration services, terminate your application and try again.

DLI_IOM_ERR_READ_CMPLT_QFULL The user's read queue is full of completed reads, which indicates the client application is not processing data received from the server fast enough. Because recognizing this condition is relatively expensive in terms of processing, it requires a DLI log level of 7 (logLev parameter on page 63). The user should examine the log file during application development.

Action: Modify the client application to service the read queues more promptly.

DLI_IOM_ERR_WRIT_CMPLT_QFULL The user's write queue is full of completed writes, which indicates the client application is not processing the completion of writes previously sent to the server fast enough. Because recognizing this condition is relatively expensive in terms of processing, it requires a DLI log level of 7 (logLev parameter on page 63). The user should examine the log file during application development.

Action: Modify the client application to service the write queues more promptly.

DLI_IOM_TSI_READ_FAILED DLI failed to issue a read request to TSI.

Action: Review the TSI error log and TSI trace file to determine possible TSI errors. Terminate your application and try again.

DLI_IOM_TSI_WRITE_FAILED DLI failed to issue a write request to TSI.

Action: Review the TSI error log and TSI trace file to determine possible TSI errors. Terminate your application and try again.

DLI_IOQU_ERR_INVALID_SESSID DLI encountered a packet from Freeway that contains an invalid session ID.

Action: Severe error; terminate your application and try again.

DLI_IOQU_ERR_IN_QFULL DLI is not able to add incoming data to the session queue because there is no room. This error only occurs when DLI is configured to handle multiple sessions per TSI connection (reuseTrans DLI configuration parameter on page 65).

Action: Revise your application logic to make sure that it does not allow one session to block incoming data to all other sessions that share the same TSI connection.

DLI_IOQU_ERR_NO_WRITES DLI encountered a completed write request from TSI that has been cancelled by your application.

Action: Revise your application logic and try again.

DLI_IOQU_ERR_QADD_FAILED DLI cannot access its internal input queue. *Action:* Severe error; terminate your application and try again.

DLI_IOQU_ERR_QEMPTY DLI internal logic error.

Action: Severe error. Terminate your application and try again.

DLI_LOGI_ERR_LOG_OPEN_FAILED DLI failed to open the log file requested through the DLI configuration file.

Action: Verify the name of the log file. Terminate your application and try again.

DLI_MEMI_ERR_CALLOC_FAILED DLI failed to allocate memory through the "calloc" function call.

Action: Severe error. Check your system configuration, terminate your application and try again. This error occurs only with the VxWorks operation system.

DLI_MEMI_ERR_SM_CREATE_FAILED DLI failed to create a shared-memory partition.

Action: Severe error. Check your system configuration, terminate your application and try again. This error occurs only with the VxWorks operation system.

DLI_MEMT_ERR_NEVER_INIT DLI memory services were never initialized.

Action: Severe error. Terminate your application and try again.

DLI_MEMT_ERR_PART_FREE_FAILED DLI failed to release the shared-memory partition.

Action: Severe error. This error occurs only with the VxWorks operation system.

DLI_RESF_MULTISQ_INVALID_NODE DLI encountered an invalid session in the multiple-session queue. This error occurs only when the multiple sessions per TSI connection option is used (reuseTrans DLI configuration parameter on page 65).

Action: Severe error; terminate your application and try again.

DLI_RESF_MULTIS_QREM_FAILED DLI failed to free an entry in the multiple-session queue. This error occurs only when the multiple sessions per TSI connection option is used.

Action: Severe error; terminate your application and try again.

DLI_RESA_ERR_BUFIO_FAILED DLI failed to allocate the internal I/O buffer for the newly created session.

Action: Severe error. Consider increasing the number of I/O buffers in the TSI configuration file. Ensure that your application releases unused buffers to DLI.

DLI_RESA_ERR_MULTIS_QADD_FAILED DLI failed to add the current session entry to the multiple-session-per-connection entry queue.

Action: Severe error. Terminate your application and try again.

DLI_RESA_ERR_MULTISQ_FAILED DLI failed to initialize its internal multiple-session-per-connection queue.

Action: Severe error. Terminate your application and try again.

DLI_RESA_ERR_NO_RESOURCE DLI failed to allocate the necessary memory resource for the internal I/O queue headers.

Action: Severe error. Terminate your application and try again.

DLI_RESA_ERR_QIN_ADD_REM_FAILED DLI failed to initialize its internal input queue. *Action:* Severe error. Terminate your application and try again.

DLI_RESA_ERR_QIO_FAILED DLI failed to initialize its internal I/O queues.

Action: Severe error. Terminate your application and try again.

DLI_RESA_ERR_QOUT_ADD_REM_FAILED DLI failed to initialize its internal output queue. *Action:* Severe error. Terminate your application and try again.

DLI_RESA_ERR_TSI_OPEN_FAILED DLI invoked tConnect to start a TSI connection and received an error.

Action: Check TSI services and its configuration. Terminate your application and try again.

DLI_SEVTP_ERR_INVALID_TRANSID DLI encountered an invalid transport ID.

Action: Severe error. Terminate your application and try again.

DLI_SINIT_ERR_CFG_LOAD_FAILED DLI failed to load the configuration entry for the specified session name. The session name is provided by your application when it invokes either a dlOpen or dlListen request. Possible errors are: invalid session name or corrupted configuration file.

Action: Review your application and re-run the dlicfg preprocessor program.

DLI_SINIT_ERR_DEQ_FAILED DLI failed to remove an inactive session from the active session queue. This error occurs only when your application issues a dlOpen or dlListen request.

Action: Severe error. Terminate your application and try again.

DLI_SINIT_ERR_GET_ENTRY_FAILED DLI failed to get a free session entry for your request.

Action: Severe error. Terminate your application and try again.

DLI_SINIT_ERR_MULTIS_QADD_FAILED DLI failed to add the current session entry to the internal multiple-session-per-connection queue.

Action: Severe error. Terminate your application and run again.

DLI_SINIT_ERR_QFULL DLI failed to accept additional dlOpen or dlListen requests because its active session queue is full.

Action: Consider increasing the maximum number of sessions allowed with DLI, terminate your application and try again.

DLI_SINIT_ERR_RESA_FAILED DLI failed to allocate the necessary system and network resources to honor your dlOpen or dlListen request.

Action: Check your system or network resources.

DLI_TRAV_ERR_INVALID_RSP DLI encountered an invalid response from Freeway or the ICP.

Action: Review the DLI error log and trace file. Terminate your application and try again.

DLI_TRAV_ERR_INVALID_STATE DLI encountered an internal logic failure.

Action: Report the error to Protogate.

DLI_TRAV_ERR_QADD_FAILED DLI could not access to its internal I/O queue.

Action: Severe error; terminate your application and try again.

DLI_TRAV_ERR_TOO_MANY_ERRORS This session has a large number of I/O errors that exceed the maximum number of errors allowed.

Action: Consider increasing the maxErrors DLI configuration parameter (page 64).

DLI_VC_ERR_ILLEGAL_SESS_TYPE The command that your application issued to Freeway or the ICP is restricted for a different protocol.

Action: Correct your application logic and try again.

B.2 Command-Specific Error Codes

Table B–1 lists alphabetically all the error codes related to specific DLI commands described in Chapter 4. These codes are returned in the global variable dlerrno.

Table B–1: DLI Command-specific Error Codes

Command(s) Causing Error	Error Code	Reference Page
dlBufAlloc	DLI_BUFA_ERR_NEVER_INIT	page 87
	DLI_BUFA_ERR_NO_BUFS	page 87
	DLI_BUFA_ERR_SIZE_EXCEEDED	page 88
dlBufFree	DLI_BUFF_ERR_INVALID_BUF	page 89
	DLI_BUFF_ERR_NEVER_INIT	page 89
	DLI_BUFF_ERR_TSI_FREE_ERR	page 89
dlClose	DLI_CLOS_ERR_FW_INVALID_RSP	page 91
	DLI_CLOS_ERR_FW_INVALID_SESS	page 91
	DLI_CLOS_ERR_FW_QADD_FAILED	page 91
	DLI_CLOS_ERR_FW_TOO_MANY_ERRORS	page 92
	DLI_CLOS_ERR_FW_UNK_STATUS	page 92
	DLI_CLOS_ERR_ICP_INVALID_RSP	page 92
	DLI_CLOS_ERR_ICP_INVALID_STATUS	page 92
	DLI_CLOS_ERR_ICP_QADD_FAILED	page 92
	DLI_CLOS_ERR_ICP_TOO_MANY_ERRORS	page 92
	DLI_CLOS_ERR_INVALID_MODE	page 92
	DLI_CLOS_ERR_INVALID_SESSID	page 93
	DLI_CLOS_ERR_INVALID_STATE	page 93
	DLI_CLOS_ERR_LINK_INVALID_RSP	page 93
	DLI_CLOS_ERR_LINK_INVALID_STATUS	page 93
	DLI_CLOS_ERR_LINK_QADD_FAILED	page 93
	DLI_CLOS_ERR_LINK_TOO_MANY_ERRORS	page 93
	DLI_CLOS_ERR_NEVER_INIT	page 93
	DLI_CLOS_ERR_Q_NOT_EMPTY	page 94
	DLI_CLOS_ERR_TOO_MANY_ERRORS	page 94

Table B–1: DLI Command-specific Error Codes (*Cont'd*)

Command(s) Causing Error	Error Code	Reference Page
dlControl	See also dlOpen	page 108
	DLI_CTRL_ERR_FAILED	page 96
	DLI_CTRL_ERR_FW_FTP_FAIL	page 96
	DLI_CTRL_ERR_FW_ICP_FAIL	page 96
	DLI_CTRL_ERR_FW_INVALID_ICP	page 97
	DLI_CTRL_ERR_FW_INVALID_RSP	page 97
	DLI_CTRL_ERR_FW_INVALID_TYPE	page 97
	DLI_CTRL_ERR_FW_SCRIPT_ERR	page 97
	DLI_CTRL_ERR_FW_UNK_STATUS	page 97
	DLI_CTRL_ERR_INIT_FAILED	page 97
	DLI_CTRL_ERR_INVALID_STATE	page 97
	DLI_CTRL_ERR_SESS_INIT_FAILED	page 98
	DLI_CTRL_ERR_TOO_MANY_ERRORS	page 98
dlClose	DLI_EWOULDBLOCK	page 91
dlControl		page 95
dlListen		page 104
dlOpen		page 108
dlRead		page 122
dlWrite		page 134

Table B–1: DLI Command-specific Error Codes (*Cont'd*)

Command(s) Causing Error	Error Code	Reference Page
dlInit	DLI_INIT_ERR_ACT_ADD_REM_FAILED	page 100
	DLI_INIT_ERR_ACT_QINIT_FAILED	page 100
	DLI_INIT_ERR_ALREADY_INIT	page 100
	DLI_INIT_ERR_CFG_LOAD_FAILED	page 100
	DLI_INIT_ERR_DLICB_ALLOC_FAILED	page 101
	DLI_INIT_ERR_GET_TSI_CFG_FAILED	page 101
	DLI_INIT_ERR_LOG_INIT_FAILED	page 101
	DLI_INIT_ERR_NAME_TOO_LONG	page 101
	DLI_INIT_ERR_NO_RESOURCE	page 101
	DLI_INIT_ERR_NO_TRACE_BUF	page 101
	DLI_INIT_ERR_TASK_VAR_FAILED	page 102
	DLI_INIT_ERR_TEXT_OPEN_FAILED	page 102
	DLI_INIT_ERR_TSI_INIT_FAILED	page 102
dlListen	DLI_LSTN_ERR_INIT_FAILED	page 105
	DLI_LSTN_ERR_INVALID_STATE	page 105
	DLI_LSTN_ERR_SESS_INIT_FAILED	page 105

Table B–1: DLI Command-specific Error Codes (*Cont'd*)

Command(s) Causing Error	Error Code	Reference Page
dlOpen	DLI_OPEN_ERR_CFG_INVALID_RSP	page 108
	DLI_OPEN_ERR_CFG_INVALID_STATUS	page 108
	DLI_OPEN_ERR_CFG_QADD_FAILED	page 109
	DLI_OPEN_ERR_CFG_TOO_MANY_ERRORS	page 109
	DLI_OPEN_ERR_FAILED	page 109
	DLI_OPEN_ERR_FW_ICP_NOT_OP	page 109
	DLI_OPEN_ERR_FW_INVALID_COMMAND	page 109
	DLI_OPEN_ERR_FW_INVALID_ICP	page 109
	DLI_OPEN_ERR_FW_INVALID_RSP	page 109
	DLI_OPEN_ERR_FW_INVALID_TYPE	page 110
	DLI_OPEN_ERR_FW_NO_SESS	page 110
	DLI_OPEN_ERR_FW_QADD_FAILED	page 110
	DLI_OPEN_ERR_FW_TOO_MANY_ERRORS	page 110
	DLI_OPEN_ERR_FW_UNK_STATUS	page 110
	DLI_OPEN_ERR_ICP_INVALID_RSP	page 110
	DLI_OPEN_ERR_ICP_INVALID_STATUS	page 110
	DLI_OPEN_ERR_ICP_QADD_FAILED	page 111
	DLI_OPEN_ERR_ICP_TOO_MANY_ERRORS	page 111
	DLI_OPEN_ERR_INIT_FAILED	page 111
	DLI_OPEN_ERR_INVALID_STATE	page 111
	DLI_OPEN_ERR_LINK_INVALID_RSP	page 111
	DLI_OPEN_ERR_LINK_INVALID_STATUS	page 111
	DLI_OPEN_ERR_LINK_QADD_FAILED	page 112
	DLI_OPEN_ERR_LINK_TOO_MANY_ERRORS	page 112
	DLI_OPEN_ERR_SESS_INIT_FAILED	page 112
	DLI_OPEN_ERR_TOO_MANY_ERRORS	page 112
dlpErrString	DLI_PRTSTRG_ERR_UNKNOWN_ERROR_NBR	page 113

Table B–1: DLI Command-specific Error Codes (*Cont'd*)

Command(s) Causing Error	Error Code	Reference Page
dlPoll	DLI_POLL_ERR_BAD_PTR	page 117
	DLI_POLL_ERR_BUF_LEN_PTR_NULL	page 117
	DLI_POLL_ERR_BUF_NOT_FOUND	page 117
	DLI_POLL_ERR_GETLIST_FAILED	page 118
	DLI_POLL_ERR_GET_TSI_CFG_FAILED	page 118
	DLI_POLL_ERR_INVALID_IOQ	page 118
	DLI_POLL_ERR_INVALID_REQ_TYPE	page 118
	DLI_POLL_ERR_INVALID_SESSID	page 118
	DLI_POLL_ERR_IO_FATAL	page 118
	DLI_POLL_ERR_NEVER_INIT	page 118
	DLI_POLL_ERR_OVERFLOW	page 119
	DLI_POLL_ERR_QEMPTY	page 119
	DLI_POLL_ERR_QREM_FAILED	page 119
	DLI_POLL_ERR_READ_ERROR	page 119
	DLI_POLL_ERR_READ_NOT_COMPLETE	page 119
	DLI_POLL_ERR_READ_QREM_FAILED	page 119
	DLI_POLL_ERR_READ_TIMEOUT	page 120
	DLI_POLL_ERR_UNBIND	page 120
	DLI_POLL_ERR_WRITE_ERROR	page 120
	DLI_POLL_ERR_WRITE_NOT_COMPLETE	page 120
	DLI_POLL_ERR_WRITE_TIMEOUT	page 120
dlPost	DLI_POST_ERR_NEVER_INIT	page 121
	DLI_POST_ERR_TSI_POST_ERR	page 121

Table B–1: DLI Command-specific Error Codes (*Cont'd*)

Command(s) Causing Error	Error Code	Reference Page
dlRead	DLI_READ_ERR_BUF_MUST_BE_NULL	page 124
	DLI_READ_ERR_INTERNAL_DLI_ERROR	page 124
	DLI_READ_ERR_INVALID_BUF	page 124
	DLI_READ_ERR_INVALID_LENGTH	page 124
	DLI_READ_ERR_INVALID_SESSID	page 125
	DLI_READ_ERR_INVALID_STATE	page 125
	DLI_READ_ERR_IO_FATAL	page 125
	DLI_READ_ERR_NEVER_INIT	page 125
	DLI_READ_ERR_OVERFLOW	page 125
	DLI_READ_ERR_QADD_FAILED	page 126
	DLI_READ_ERR_QFULL	page 126
	DLI_READ_ERR_READ_ERROR	page 126
	DLI_READ_ERR_TIMEOUT	page 126
	DLI_READ_ERR_TOO_MANY_ERRORS	page 126
	DLI_READ_ERR_TSI_BUFF_MISSING	page 126
	DLI_READ_ERR_UNBIND	page 126
dlSyncSelect	DLI_SYNCSELECT_ERR_INVALID_ARRAY	page 129
	DLI_SYNCSELECT_ERR_INVALID_SESSID	page 130
	DLI_SYNCSELECT_ERR_INVALID_STATE	page 130
	DLI_SYNCSELECT_ERR_NEVER_INIT	page 130
	DLI_SYNCSELECT_ERR_NOT_SYNC	page 130
	DLI_SYNCSELECT_ERR_TSI_ERROR	page 130
dlTerm	DLI_TERM_ERR_ACT_REM_FAILED	page 133
	DLI_TERM_ERR_ACT_TERM_FAILED	page 133
	DLI_TERM_ERR_CLOSE_FAILED	page 133
	DLI_TERM_ERR_LOG_END_FAILED	page 133
	DLI_TERM_ERR_NEVER_INIT	page 133
	DLI_TERM_ERR_RES_FREE_FAILED	page 133
	DLI_TERM_ERR_TSI_TERM_FAILED	page 133

Table B–1: DLI Command-specific Error Codes (*Cont'd*)

Command(s) Causing Error	Error Code	Reference Page
dlWrite	DLI_WRIT_ERR_BUFA_FAILED	page 137
	DLI_WRIT_ERR_ILLEGAL_ICP_PROT_CMD	page 137
	DLI_WRIT_ERR_ILLEGAL_SERVER_CMD	page 137
	DLI_WRIT_ERR_INTERNAL_DLI_ERROR	page 137
	DLI_WRIT_ERR_INVALID_BUF	page 137
	DLI_WRIT_ERR_INVALID_LENGTH	page 137
	DLI_WRIT_ERR_INVALID_SESSID	page 138
	DLI_WRIT_ERR_INVALID_STATE	page 138
	DLI_WRIT_ERR_INVALID_WRITE_TYPE	page 138
	DLI_WRIT_ERR_IO_FATAL	page 138
	DLI_WRIT_ERR_LOCAL_ACK_ERROR	page 138
	DLI_WRIT_ERR_NEVER_INIT	page 138
	DLI_WRIT_ERR_QADD_FAILED	page 138
	DLI_WRIT_ERR_QFULL	page 139
	DLI_WRIT_ERR_TIMEOUT	page 139
	DLI_WRIT_ERR_TOO_MANY_ERRORS	page 139
	DLI_WRIT_ERR_UNBIND	page 139
	DLI_WRIT_ERR_WRITE_ERROR	page 139

B.3 Error Handling for Dead Socket Detection

A catastrophic I/O failure between the client and server generates a “dead-socket” condition. This condition can be recognized by the DLI application through DLI’s session status and from the error returned with the I/O buffer whose operation detected the failure. In most cases of catastrophic failure, TSI closes the client-server connection. However, when the DLI application is notified of a dead socket, no assumptions should be made regarding the current state of the connection; in all cases the DLI session should be closed.

Dead sockets change the DLI session status to DLI_STATUS_DEAD_SOCKET (returned from a dlPoll request with the DLI_POLL_GET_SESS_STATUS option). If the application uses blocking I/O, the I/O request is returned the DLI_…_ERR_IO_FATAL error. For non-blocking I/O, the I/O request which detected the failure returns the DLI_…_ERR_IO_FATAL error, and all pending I/O operations which have not been completed are returned with the DLI_…_ERR_UNBIND error.

In a dead-socket condition, the DLI session remains open until closed the by application. However, DLI does not allow the application to perform any read or write requests (all requests are returned with an …_INVALID_STATE error). The application can retrieve any outstanding I/O requests by using dlPoll to request read or write completions. Requests which were completed before the dead-socket condition occurred are returned with their appropriate status. However, write buffers awaiting Local Acks and read requests not yet performed are returned with the …UNBIND error code.

The application must close the DLI session. While the session is in the dead-socket condition, dlPoll requests are allowed, and the session can be closed, but all other requests are returned with an error indication. Session resources are retained until the session is closed by the application. The application should not assume callbacks from dlClose (TSI may have closed the client-server connection). Additionally, errors from the dlClose request might be considered normal since DLI will attempt to close the TSI connection regardless of the connection's current state. DLI forces the close processing regardless of TSI's response to a DLI close request.

The TSI recognizes a dead socket by a failure in a read or write attempt. While writes rarely return errors, they are required to recognize the dead socket condition after the socket is down. Specifically, the application must issue a write to recognize a dead-socket condition. In applications using non-blocking I/O, a read request must be pending.

Appendix C

UNIX, VxWorks, and VMS I/O

C.1 UNIX Environment

The DLI provides the non-blocking I/O operation through the services provided by the TSI layer. The TSI interacts directly with the UNIX system services to gain access to non-blocking I/O services through the use of a signal delivery mechanism. When a signal is delivered to TSI through the use of an interrupt service routine, TSI immediately suspends the delivery of that signal again until it completes its I/O services through the IOCH function. TSI will exit the IOCH either when it runs out of system resources to accept additional I/O, or when it has no additional I/O to accept. In either case, system resources will be tied up by TSI while it is in the IOCH function unless it is interrupted by another system service request (i.e. another signal delivery) with a higher priority than its own. When TSI completes its own I/O services, it will invoke the DLI IOCH. If your application decides to use non-blocking I/O and provides DLI with an IOCH, the DLI IOCH will subsequently invoke your application IOCH after it completes its I/O services. As you see, there are three levels of IOCH that are invoked to complete an I/O condition.

In short, non-blocking I/O operation is not only complex but also expensive. Therefore, it requires careful planning and design so that your application uses the system resources wisely.

Note

Also see Section 2.5.2 on page 53 for more information on signal processing.

C.1.1 Blocking I/O Operations

Blocking I/O operation requires no IOCHs. Blocking I/O does not use any signal delivery mechanism to handle the delivery of data. Blocking I/O allows the orderly execution of your application and requires far fewer system resources than non-blocking I/O. If you design a DLI application to interact with a remote data link application, you should consider the blocking I/O feature. Blocking I/O is also easier to debug and troubleshoot than non-blocking I/O. Careful design through the isolation of system and protocol dependency will allow your application to work not only in blocking mode but also in non-blocking mode. The DLI and TSI services allow your application to switch from blocking mode to non-blocking mode, and vice versa, without the recompilation of your application code.

It is difficult to handle multiple sessions under blocking I/O operation, because your application will be blocked until the data arrives or DLI times out while waiting. While your application is waiting for I/O in one session, data from other sessions is blocked.

C.1.2 Non-blocking I/O Operations

The DLI uses the SIGIO signal for its non-blocking BSD socket interface. Therefore, your application should not block the delivery of SIGIO signals (for example, sigprocmask) at any time, especially when expecting data from the network.

If you use non-blocking I/O, design your application with robust IOCH function(s). Also, the application IOCH should perform as little work as possible and before it exits, use some notification techniques to awaken the main routines to perform the remaining tasks. Some possible notification techniques are system semaphores, sleep and wakeup calls using the SIGALRM signal, etc.

C.1.3 SOLARIS use of SIGALRM

The use of a default signal handler through SIGALRM signal can cause a system core dump inside the SOLARIS internal SIGALRM signal handler. You can work around it by providing your own signal handler for SIGALRM. The following code segment assists you in setting up a SIGALRM handler for the SIGALRM signal:

```
void genSigHandlr ( int signal )
{
        return;
}

void main ( )
{
        struct sigaction            SigAction;

        SigAction. sa_handler = genSigHdlr;
        sigfillset (&SigAction. sa_mask);
        SigAction. sa_flags = 0;

        if (sigaction (SIGALRM, &SigAction, (struct sigaction *)NULL) ==
                ERROR)
        {
                fprintf (stderr, "sigaction failed %d\n", errno);
                return ERROR;
        }
        .....
        return OK;
}
```

Notice that genSigHdlr does nothing but return to the system.

C.1.4 Polling I/O Operations

Your application can implement polling I/O operations if it uses DLI with non-blocking I/O but provides no IOCH functions. Since your application provides no mechanism for DLI to notify it when an I/O condition occurs, your application must poll DLI for the completion of I/O requests that it posts to DLI. Polling I/O operations involve the dlPoll function (Section 4.10 on page 114). Polling I/O is helpful if your application manages multiple sessions, data arrives at a predictable rate, and the timing of data is not critical.

C.2 VxWorks Environment

DLI and TSI will operate only in a VxWorks environment that is similar to that of the Freeway server. VxWorks has several features similar to UNIX; however, it has a unique operating environment and a real-time operating system. The use of DLI and TSI together by an application that runs on the Freeway server is often called a server-resident application (SRA). The SRA can be configured to interact with Protogate's message multiplexor subsystem through the shared-memory transport mechanism supported by TSI, or it can be configured to interact with other systems using the BSD socket interface which is also supported by TSI. Whichever transport your SRA program uses, you should understand not only the VxWorks operating system but also the way the Freeway server is configured and how Protogate implements TSI and DLI under VxWorks. For more information on SRAs, see the *Freeway Server-Resident Application and Server Toolkit Programmer Guide*.

C.2.1 Blocking I/O Operations

Blocking I/O in VxWorks is similar to that of the UNIX environment.

C.2.2 Non-blocking I/O Operations

Non-blocking I/O in VxWorks with Protogate's Freeway server requires your application to cooperate with other tasks. VxWorks on Freeway is configured to operate in a cooperative manner. This means that VxWorks operates as a non-preemptive multitasking environment. When your application does not have data to be processed, it must relinquish the CPU so that other tasks can run. You can use your own interrupt service routine to notify or resume your application when its data arrives.

The DLI uses a binary semaphore to support non-blocking I/O delivery from both network and shared-memory environments. Since VxWorks running on Freeway is configured for a cooperative environment, your application must also act cooperatively. Your application must call dlPost immediately before relinquishing its control to VxWorks. Your application must relinquish the control through taskDelay, binary semaphores, or other means; otherwise, only your task has control of the CPU which prevents other important tasks from running.

Your application should use global variables sparingly if multiple instances of the same application are running concurrently. VxWorks global variables are shared among all tasks unless you define them as a particular task's variables (using taskVarAdd). Task variables are expensive to maintain by VxWorks and therefore should be used sparingly.

C.3 VMS Environment

The DLI uses the process-level Asynchronous System Trap (AST) for non-blocking data delivery from the network. Therefore, your application should not block the delivery of ASTs (using sys$setast) at any time, especially when expecting data from the network.

Appendix

D DLI Logging and Tracing

In conjunction with the transport subsystem interface (TSI), DLI provides tracing and logging services to troubleshoot both application and network problems. Both logging and tracing services are included in DLI and TSI. Refer to the *Freeway Transport Subsystem Interface Reference Guide* for more information on TSI logging and tracing.

D.1 DLI Logging

There are two kinds of DLI logging services: general logging and session-related logging. As the name implies, general logging includes errors or information not related to any particular session. Session-related logging indicates error or information related to a specific session. To monitor data, you must use the DLI tracing services described in Section D.2.

General logging is defined in the "main" section of the DLI text configuration file. The logLev parameter (page 63) specifies the level of logging your application needs and can be from 0 to 7, with level 0 being no logging, level 1 being the most severe error, and 7 being the least severe. In the "main" section, the logName parameter (page 63) defines the log file name where your logging information is to reside. The default file name is "dlilog." If you wish logging information to be output to the screen, define logName as "stdout." The number of entries to "stdout" is unlimited. A disk file is limited to 1000 entries, and this number is not configurable.

Session-related logging can be defined in each individual DLI session definition. You can log for some sessions but not for the others; and different sessions can log errors at different levels. All error codes are defined in Appendix B and in each individual function description (for example, dlOpen in Section 4.8 on page 106).

The following is the format of the each log entry:

SessX: DLI_YYY_ZZZ_Information(dlerrno/errno)

where:

X is session ID. For general logging, X will be 999. Otherwise, it indicates a session-related entry.

YYY is brief function name of DLI. For example, if ZZZ is OPEN it indicates the log entry is from dlOpen function.

ZZZ can be ERR or INFO. ERR indicates an error condition; INFO indicates information only.

dlerrno is a DLI error code for this entry; errno is the last encountered 'C' errno value.

D.2 DLI Tracing

D.2.1 Trace Definitions

The DLI tracing facility captures and stores real-time data in its internal wrap-around buffer. The size of this buffer is configurable up to 1 megabyte of memory. There are two kinds of DLI tracing: general tracing and session-specific tracing. In general tracing, trace data has no session-specific information, whereas session-specific trace data pertains to only one specific session ID.

To activate tracing, first specify the DLI "main" configuration parameters. Specify the traceSize parameter (page 63) up to 1 megabyte of memory. The traceName parameter (page 63) defines the file name where your trace information is to reside.

Specify the level of tracing using the traceLev parameter (page 63). This parameter defaults to zero if not defined (no tracing). The traceLev parameter can be defined in the "main" section for general tracing or in each individual session definition (page 65). Each session definition can have different traceLev value.

The traceLev parameters can be the sum of one or more of the following values:

1 = trace the read (input) data

2 = trace the write (output) data

4 = trace the DLI interrupt services

8 = trace the application IOCH services

16 = trace the user's data

For example, if you want to trace both read and write data, specify 3 for the traceLev parameter. If you want to trace read, write, and user's data, specify 19 for the traceLev parameter.

The most commonly used trace level is for I/O passing through the DLI service layer (traceLev = 3). DLI also provides the interrupt and application I/O completion handler (IOCH) trace levels within DLI to assist the application in troubleshooting the IOCH mechanism. The user data trace level allows the application to store its own data in the trace buffer.

Note

DLI does not decode user data with its dlidecode program (Section D.2.2).

You can turn tracing *on* or *off* at any time after DLI is initialized using dlPoll with the DLI_POLL_TRACE_ON or DLI_POLL_TRACE_OFF options. Tracing is done internally with the DLI trace buffer. Trace data is not written to the trace file until dlTerm is called or dlPoll is called with the DLI_POLL_TRACE_WRITE option. Therefore, your application should always call dlTerm before it exits to the operating system. If tracing is required and is defined in the DLI configuration file, it is automatically *on* when dlInit is called. You can use dlPoll with the DLI_POLL_TRACE_STORE option to store your own trace buffer inside the DLI trace buffer. Refer to dlPoll (Section 4.10 on page 114) for more information. Since DLI tracing does not involve disk I/O, there is little or no performance impact.

D.2.2 Decoded Trace Layout

You can run the dlidecode program against the trace file produced by DLI. The output of dlidecode is output to the screen for UNIX-like systems. In VMS, dlidecode's output is output to the file named dli.sum. You can also run dlidecode against the trace file produced by TSI. Refer to the *Freeway Transport Subsystem Interface Reference Guide* for details on TSI tracing.

The format of the decoded trace can be described as follows. See Section D.2.3 for an actual decoded trace example.

```
line 1: Protogate 2000(C) DLI Trace Decoder
line 2: Max buffer size: xyz
line 3: TRACE SOURCE: yyy -----------------------------------------------------------
line 4: @@@@@ Actual Data offset aa Size = bb
line 5: cc: hex data and printable ascii equivalent.
line 6: @@@@@ Decode begins
line 7: dd(desc) Conn ee: time and date
line 8: Freeway header info: length = ff
line 9: Packet Type (gg) = textgg Command(hh) = texthh
line 10: Status(ii) = textii Client ID = jj Freeway ID = kk
line 11: ICP header info:
line 12: OldClientID = ll OldServerID = mm
line 13: Data length = nn Cmd (oo) = textoo
line 14: Status(pp) = textpp
line 15: Parms: [0] = qq [1] = rr [2] == ss
line 16: Protocol header info:
line 17: Cmd(tt) = texttt
line 18: Modifier = uu Link = vv
```

line 19: Cir = xx Sess = yy Seq = zz
line 20: Parms: [0] = a1 [1] = b1
line 21: DATA : hex data and printable ascii equivalent.

Each line of the above format is explained as follows:

line 1: indicates the copyright and the name of the dlidecode program. Note that TSI has its own decoder (tsidecode) which can run only against the TSI trace file, unlike dlidecode which can run on both DLI and TSI trace files.

line 2: prints the currently used maximum buffer size that is defined in the TSI configuration file (maxBufSize on page 148). Note that the size excludes the overhead used by DLI and TSI; it describes the maximum number of actual data bytes allowed by DLI. See Section 2.4 on page 40.

line 3: prints the source of the trace.

line 4: describes the actual offset (aa) from the beginning of the trace file where this packet is stored and the number of bytes contained in this packet has (bb). Section D.2.4 describes how to read the DLI trace file in case you want to write your own decoder to decode your own trace data that you store in DLI trace buffer using dlPoll with the DLI_POLL_TRACE_STORE option.

line 5: prints the actual hex values and their equivalent printable ASCII text. The offset (cc) is the actual offset from the beginning of the packet, based on 0. Each line contains up to 16 bytes from the trace packet. Note that line 5 can be repeated if the actual size of the trace packet is more than 16 bytes long.

line 6: indicates that the actual decoding begins. This is where the headers are broken into individual fields.

line 7: dd indicates the direction of the packet; dd can be ====> to indicate an outgoing packet or <==== to indicate an incoming packet. For non-I/O related packets (for example, user's data packet), dd is either ***** or #####. If the trace packet is for I/O, desc can be READ(1)/WRITE(2) n bytes. If the packet is a non-I/O packet, desc can be one of the following:

SESSION INTERRUPT BEGINS(4): indicates that DLI begins its interrupt handler to process I/O requests.

SESSION INTERRUPT ENDS(5): indicates that DLI ends its interrupt handler routine and is ready to return to TSI.

SESSION ISR(3): indicates that DLI is about to call the IOCH of a specific session ID. The address of this IOCH was provided by TSI to DLI through the dlOpen function.

APPLICATION ISR BEGINS(6): indicates that DLI is about to call the generic IOCH that was provided by the application to DLI through the dlInit function.

APPLICATION ISR ENDS(7): indicates that the generic IOCH routine returns control to DLI.

AT(8): indicates that this trace buffer belongs to the application. The dlidecode program will not attempt to decode this packet. You have to write your own decode function to interpret your own data packet.

line 8: begins the Freeway header information. The length of the Freeway header is also printed (ff).

line 9: describes the packet type of this Freeway packet. There are two types of packets supported by Freeway: the FW_CONTROL control packet and the FW_DATA data packet. Any other packet type is rejected by Freeway.

textgg is the English version of the packet type.

hh indicates the command that was issued to or from Freeway.

texthh describes the command in English.

line 10: ii is the status value returned by Freeway. If the packet is outgoing, this field contains an internal value used by DLI and has no meaning to the application. jj is the client ID provided by DLI. This client ID is the same as the client ID returned from the dlOpen function. kk is the Freeway ID that is assigned and returned by Freeway.

Note

Note that line 11 through line 20 might not be included if the packet destination is the Freeway server only. If the packet is targeted to the ICPs, line 11 through line 20 will be included. The fields of the ICP and Protocol headers are described briefly below. If you need further information about these headers, refer to your particular protocol programmer's guide.

line 11: indicates the beginning of the ICP header.

line 12: ll and mm are the values of the old client ID. It is used only by the X.25 protocol.

line 13: nn indicates the length of the data area plus the length of the Protocol header; oo is the value of the command to the ICP; and textoo is the command's name in English.

line 14: pp indicates the error status from the ICP or contains a value used internally by DLI to indicate to the ICP the host's machine architecture (Big-Endian versus Little-Endian).

line 15: prints the 3 values of the extra parameters in the ICP header.

lines 16 through 20: describe the protocol-specific information. Refer to your particular protocol programmer's guide for further information.

line 21: prints the details of the data in both hex values and the printable ASCII equivalent.

D.2.3 Example dlidecode Program Output

Following are example segments of the actual output from the dlidecode program:

```
Protogate 2000(C) DLI Trace Decoder

Max buffer size: 436
TRACE SOURCE: DLI

------------------------------------------------------------------------------

@@@@@ Actual Data offset 8 Size = 0
@@@@@ Decoding begins

                DATA  : 00 00 00 00 00 00 00 00 2e a8 3d 6a          ..........=j


------------------------------------------------------------------------------

@@@@@ Actual Data offset 20 Size = 76
                000000: 00 00 00 00 00 00 00 00 00 0a 00 16 00 01 00 01  ................
                000016: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
                000032: 00 00 00 00 00 00 00 00 00 00 00 00 69 63 70 30  ............icp0
                000048: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
                000064: 00 00 00 00 00 00 00 00 00 00 00 00              ............
@@@@@ Decoding begins

====>(WRITE 76 bytes)Conn 0: Fri Oct 21 15:15:07 1994
                Freeway header info: length = 44
                Packet Type(1) = FW_CONTROL Command(1) = FW_OPEN_SESS
                Status(1) = INV_ICP Client ID = 0 Freeway ID = 0

                DATA  : 69 63 70 30 00 00 00 00 00 00 00 00 00 00 00 00  icp0............
                DATA  : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

------------------------------------------------------------------------------

@@@@@ Actual Data offset 108 Size = 76
                000000: 00 00 00 00 00 00 00 00 00 0a 00 16 00 01 00 01  ................
                000016: 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00  ................
                000032: 00 00 00 00 00 00 00 00 00 00 00 00 69 63 70 30  ............icp0
                000048: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
                000064: 00 00 00 00 00 00 00 00 00 00 00 00              ............
@@@@@ Decoding begins

====>(WRITE 76 bytes)Conn 1: Fri Oct 21 15:15:07 1994
```

```
                Freeway header info: length = 44
                Packet Type(1) = FW_CONTROL Command(1) = FW_OPEN_SESS
                Status(1) = INV_ICP Client ID = 1 Freeway ID = 0

                DATA  : 69 63 70 30 00 00 00 00 00 00 00 00 00 00 00 00  icp0............
                DATA  : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

------------------------------------------------------------------------------

@@@@@ Actual Data offset 196 Size = 64
                000000: 00 00 00 00 00 00 00 00 00 0a 00 16 00 01 00 01  ................
                000016: 00 00 00 00 00 03 00 00 00 00 00 00 00 00 00 00  ................
                000032: 00 00 00 14 00 00 00 00 00 03 00 00 46 72 65 65  ............Free
                000048: 77 61 79 20 52 65 6c 65 61 73 65 20 32 2e 30 00  way Release 2.0.
@@@@@ Decoding begins

<====(READ 64 bytes)Conn 0: Fri Oct 21 15:15:07 1994
                Freeway header info: length = 44
                Packet Type(1) = FW_CONTROL Command(1) = FW_OPEN_SESS
                Status(0) = OK Client ID = 0 Freeway ID = 3

                DATA  : 46 72 65 65 77 61 79 20 52 65 6c 65 61 73 65 20  Freeway Release
                DATA  : 32 2e 30 00                                      2.0.


------------------------------------------------------------------------------

@@@@@ Actual Data offset 272 Size = 76
                000000: 00 00 00 00 00 00 00 00 00 0a 00 16 00 02 00 01  ................
                000016: 00 02 00 00 00 03 00 00 00 00 00 00 00 00 00 00  ................
                000032: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
                000048: 00 10 08 01 00 00 00 00 00 00 00 00 08 01 00 01  ................
                000064: 00 00 00 00 00 00 00 00 00 1c 00 00              ............
@@@@@ Decoding begins

====>(WRITE 76 bytes)Conn 0: Fri Oct 21 15:15:07 1994
                Freeway header info: length = 44
                Packet Type(2) = FW_DATA Command(1) = FW_ICP_WRITE
                Status(2) = ICP_NOT_OP Client ID = 0 Freeway ID = 3

                        ICP header info:
                        OldClientID = 0 OldServerID = 0
                        Data length = 16 Cmd(2049) = DLI_ICP_CMD_ATTACH
                        Status(0) = DLI_ICP_ERR_NO_ERR
                        Parms: [0] = 0 [1] = 0 [2] = 0

                        Protocol header info:
                        Cmd(2049) = DLI_ICP_CMD_ATTACH
                        Modifier = 1 Link = 0
                        Cir = 0 Sess = 0 Seq = 0
                        Parms: [0] = 28 [1] = 0
```

```
-------------------------------------------------------------------------------

@@@@@ Actual Data offset 360 Size = 64
                000000: 00 00 00 00 00 00 00 00 00 0a 00 16 00 01 00 01  ................
                000016: 00 00 00 01 00 04 00 00 00 00 00 00 00 00 00 00  ................
                000032: 00 00 00 14 00 00 00 00 00 04 00 00 46 72 65 65  ............Free
                000048: 77 61 79 20 52 65 6c 65 61 73 65 20 32 2e 30 00  way Release 2.0.
@@@@@ Decoding begins

<====(READ 64 bytes)Conn 1: Fri Oct 21 15:15:07 1994
                Freeway header info: length = 44
                Packet Type(1) = FW_CONTROL Command(1) = FW_OPEN_SESS
                Status(0) = OK Client ID = 1 Freeway ID = 4

                DATA  : 46 72 65 65 77 61 79 20 52 65 6c 65 61 73 65 20  Freeway Release
                DATA  : 32 2e 30 00                                      2.0.


-------------------------------------------------------------------------------

@@@@@ Actual Data offset 436 Size = 76
                000000: 00 00 00 00 00 00 00 00 00 0a 00 16 00 02 00 01  ................
                000016: 00 02 00 01 00 04 00 00 00 00 00 00 00 00 00 00  ................
                000032: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
                000048: 00 10 08 01 00 00 00 00 00 01 00 00 08 01 00 01  ................
                000064: 00 01 00 00 00 00 00 00 00 1c 00 00              ............
@@@@@ Decoding begins

====>(WRITE 76 bytes)Conn 1: Fri Oct 21 15:15:07 1994
                Freeway header info: length = 44
                Packet Type(2) = FW_DATA Command(1) = FW_ICP_WRITE
                Status(2) = ICP_NOT_OP Client ID = 1 Freeway ID = 4

                        ICP header info:
                        OldClientID = 0 OldServerID = 0
                        Data length = 16 Cmd(2049) = DLI_ICP_CMD_ATTACH
                        Status(0) = DLI_ICP_ERR_NO_ERR
                        Parms: [0] = 0 [1] = 1 [2] = 0

                        Protocol header info:
                        Cmd(2049) = DLI_ICP_CMD_ATTACH
                        Modifier = 1 Link = 1
                        Cir = 0 Sess = 0 Seq = 0
                        Parms: [0] = 28 [1] = 0


-------------------------------------------------------------------------------
```

D.2.4 Trace Binary Format

You can use the following information to write your own decoder if you need to provide your own trace information. The trace file format is shown in Figure D–1.

TRACE_FCB
DLI_TRACE_HDR
TRACE_PACKET
DLI_TRACE_HDR
TRACE_PACKET
⋮
DLI_TRACE_HDR
TRACE_PACKET

Figure D–1: DLI Trace File Format

Figure D–2 shows the TRACE_FCB ‘C’ structure.

```
typedef struct          _TRACE_FCB
{
            int         iMaxBufSize;
            short       iTraceSource;
            short       iPadding;
}                       TRACE_FCB, *PTRACE_FCB;
```

Figure D–2: TRACE_FCB ‘C’ Structure

Figure D–3 shows the format of each trace packet in the DLI_TRACE_HDR ‘C’ structure.

```
typedef struct      _DLI_TRACE_HDR
{
    unsigned short  usTrcType;      /* type of tracing          */
    unsigned short  usTrcSessID;    /* current session ID       */
    int             iTrcDataSize;   /* sizeof the trace packet  */
    time_t          tTrcTime        /* time stamp               */
}                   DLI_TRACE_HDR, *PDLI_TRACE_HDR;
```

Figure D–3: DLI_TRACE_HDR “C” Structure

D.3 Freeway Server Tracing

Tracing service is also provided from the Freeway server. Refer to the *Freeway User Guide* for more information. You can use the trace information from both the client application and the Freeway server to diagnose and troubleshoot your client application. The Freeway trace service is identical to that of the client application; however, the direction of the trace is the reverse of that of the client. For example, for the same data packet, the client would indicate a read packet while the server would indicate a write packet. Care therefore must be taken when translating the two traces.

Index

A
Acknowledgment
  see Data acknowledgment
Addressing
  Internet 27
alwaysQIO DLI parameter 33, 64, 164
Asynchronous I/O
  see Non-blocking I/O
asyncIO DLI parameter 33, 63, 64, 162
asyncIO TSI parameter 148, 149
Audience 13

B
Binary configuration files 27, 32, 56, 58, 59, 60, 62, 96, 99, 103, 107, 150, 161
  management 60
Bit numbering 17
Blocking I/O 32, 104, 108
  caution 115
  code example 154
  DLI configuration 144, 157
  DLI session status 81, 115
  example program 144
  signal processing 53
  TSI configuration 146
  UNIX 206
  VxWorks 208
boardNo DLI parameter 64, 82
BSC msgBlkSize parameter 41, 42
Buffer management 40
  allocation and release 50
  buffer size negotiation 49
  cautions 51
  client buffers 41
  client configuration 45
  connection-specific buffers 48
  example calculation 42
  headers 51
  ICP buffers 41
  malloc vs dlBufAlloc 52
  muxCfg file 51
  server buffers 41
  TSI buffer pool definition 46
  using your own buffers 51
Byte ordering 17

C
callbackQsize DLI parameter 63
Categories of DLI functions 74
Caution
  blocking I/O 115
  data loss 75
  dlOpen completion status 106
  dlPoll session status 80
  signal processing 53
cfgLink DLI parameter 64, 106, 150, 163
Client buffer configuration 45
Client buffers 41
  allocation and release 50
  connection-specific 48
Client operations 27
Client-server environment 26
  establishing Internet address 27
Configuration 31
  binary files 32, 56, 58, 59, 60, 62, 96, 99, 103, 107, 150, 161
    management 60
  client buffers 45
  data link 36
  definition parameters

protocol specific 66
DLI 55
alwaysQIO parameter 33, 64, 164
asyncIO parameter 33, 63, 64, 162
blocking I/O 144, 157
boardNo parameter 64, 82
callbackQsize parameter 63
cfgLink parameter 64, 106, 150, 163
client parameters 64
enable parameter 64, 106, 150, 163
family parameter 64, 70
localAck parameter 64, 134, 151, 164
logLev parameter 63, 64, 211
logName parameter 63, 211
main parameters 63
main section 62
maxErrors parameter 64, 81, 112, 126, 139, 193
maxInQ parameter 53, 64, 81
maxOutQ parameter 53, 65, 81
maxSess parameter 53, 63, 79, 104, 107
mode parameter 65, 70, 82, 145, 158
msgBlkSize parameter 66
portNo parameter 65, 82
protocol parameter 34, 38, 65, 70, 106, 145, 158
protocol-specific parameters 66
reuseTrans parameter 65, 190, 191
sessions 62
sessPerConn parameter 63
summary 55
text file 62
timeout parameter 103
traceLev parameter 63, 65, 213
traceName parameter 63, 116, 212
traceSize parameter 63, 101, 116, 212
transport parameter 35, 39, 62, 65, 146
tsiCfgName parameter 62, 63
writeType parameter 34, 66, 134, 151, 163
DLI and TSI 27
dlicfg program 32, 56, 99
error messages 69
file
DLI example 145, 158
example 67
rules 59
TSI example 147, 160
grammar (PDL) 71
language 59
on-line processing 61
session definition parameters 61
system 47, 78
TSI
asyncIO parameter 148, 149
blocking I/O 146
connection parameters 149
logLev parameter 148, 149
logName parameter 148
main parameters 148
maxBuffers parameter 46, 51, 53, 79, 86, 148
maxBufSize parameter 42, 46, 47, 48, 49, 50, 51, 53, 119, 125, 148, 149, 215
maxConn parameter 148
non-blocking I/O 159
server parameter 149
summary 56
timeout parameter 53, 120, 126, 139, 149
traceLev parameter 148, 149
traceName parameter 148
traceSize parameter 148
transport parameter 149
wellKnownPort parameter 149
tsicfg program 32, 56
Configuration processor
see dlicfg preprocessor program
Connection
TSI 38
TSI configuration 35, 39, 56, 62
example 147, 160
TSI termination 37
Connection status 49, 182
Connection-specific buffers 48
Context free grammar 71
Customer support 19

D

Data
caution, data loss 75
exchanging with remote application 28

format 83
headers 83
Data acknowledgment 64, 151
see also localAck DLI parameter
Data link interface
see DLI
Data link interface (DLI) 26, 27
Data structures 78
protocol optional arguments 83
session status 80, 114, 115
system configuration 78, 115
Data transfer 37
dlRead 122
dlWrite 134
Dead socket status 202
Dead sockets 202
error handling 202
TSI recognition 203
Direct memory access 26
dlBufAlloc *(see also* Functions) 86
dlBufFree *(see also* Functions) 89
dlClose *(see also* Functions) 90
dlControl *(see also* Functions) 95
dlerrno global variable 73, 76, 113, 187, 195
DLI
configuration 55
blocking I/O 157
error codes 187
features 28
function categories 74
functions 73, 75
see also Functions
header 48, 51
header files 185
overview 28
see also Configuration, DLI
DLI configuration parameters
BSC msgBlkSize 41, 42
DLI functions
overview 74
syntax synopsis 76
dlicfg preprocessor program 32, 56, 99
error messages 69
grammar 71
introduction 58
language 59
rules 59
dlierr.h include file 187
dlInit *(see also* Functions) 99
DLITE embedded interface 22
dlListen *(see also* Functions) 103
dlOpen *(see also* Functions) 106
dlpErrString *(see also* Functions) 113
dlPoll *(see also* Functions) 114
dlPost *(see also* Functions) 121
dlRead *(see also* Functions) 122
dlSyncSelect *(see also* Functions) 128
dlTerm *(see also* Functions) 132
dlWrite *(see also* Functions) 134
Documents
reference 14
Download software 27, 74, 75
using dlControl 95

E

Electrical interface 25
Embedded ICP
environment 27
overview 22
Embedded ICP2432 30
enable DLI parameter 64, 106, 150, 163
Error codes 187
…_INVALID_STATE 202
…INVALID_STATE 81, 90
…UNBIND 202
command-specific 195
dlerrno global variable 73, 76, 113, 152, 164, 165, 187, 195
DLI_…_ERR_IO_FATAL 202
DLI_…_ERR_UNBIND 202
DLI_EWOULDBLOCK 196
DLI_POLL_ERR_BUF_LEN_PTR_NULL 117
DLI_READ_ERR_OVERFLOW 123
internal 187
see also Functions (return codes listed under each function)
TSI_READ_ERR_OVERFLOW 51
TSI_WRIT_ERR_INVALID_LENGTH 49, 51
Error handling

dead sockets 202
DLI logging 211
Error messages
dlicfg 69
Errors
dead socket detection 202
Ethernet 25
Example program
blocking I/O 144
dlControl 179
dlPoll 182
non-blocking I/O 157
raw operation 176
Examples
calculation of buffer sizes 42
DLI configuration file 67
blocking I/O 145
non-blocking I/O 158
trace output 218
TSI configuration file
blocking I/O 147
non-blocking I/O 160
tutorial programs 141

F

family DLI parameter 64, 70
Features
DLI 28
ICP2432 embedded environment 30
product 25
Files
binary configuration 56
management 60
dlierr.h include file 187
example DLI configuration 67
blocking I/O 145
non-blocking I/O 158
example TSI configuration
blocking I/O 147
non-blocking I/O 160
fmpasdcfg DLI configuration 158
fmpastcfg TSI configuration 160
fmpssdcfg DLI configuration 145
fmpsstcfg TSI configuration 147
header include files 185
muxCfg 51
on-line configuration 61
fmpasdcfg DLI configuration file 158
fmpastcfg TSI configuration file 160
fmpssdcfg DLI configuration file 145
fmpsstcfg TSI configuration file 147
Freeway
client-server environment 26
header 83
overview 22
Functions
buffer management
dlBufAlloc 86
return codes 87, 195
see also 76, 142
dlBufFree 89
return codes 89, 195
see also 77, 142
control functions
dlControl 95
example 179
return codes 96
see also 77
data transfer
dlpErrString 73, 113
return codes 198
see also 76
dlPoll 114
DLI_POLL_GET_SESS_STATUS
option 80, 104, 108, 123, 202
DLI_POLL_GET_SYS_CFG
option 78, 86, 87, 125, 135, 137
example 182
return codes 117, 199
see also 76, 143, 162, 165
dlPost 121
return codes 113, 121, 199
dlRead 122
return codes 123, 200
see also 76, 142, 152, 164, 165, 174
dlSyncSelect 128
return codes 129, 200
see also 77
dlWrite 134

return codes 136, 201
see also 76, 142, 151, 163, 164, 174
DLI preparation
dlInit 99
return codes 100, 197
see also 76, 142, 150, 161
dlTerm 132
return codes 132, 200
see also 76, 77, 142, 153, 166
dlPoll
DLI_POLL_GET_SESS_STATUS
option 47, 49, 106
example program 141, 182
session handling
dlClose 90
return codes 91, 195
see also 77, 142, 153, 165
dlListen 103
return codes 104, 197
see also 76
dlOpen 106
caution, completion status 106
return codes 107, 198
see also 76, 142, 150, 162
tBufAlloc 50
tBufFree 50
tPoll
TSI_POLL_GET_CONN_STATUS
option 49, 182
TSI_POLL_GET_SYS_CFG option 47

G

Grammar
context free 71
PDL 71

H

Headers
data format 83
DLI 48, 51
example 48
files 185
Freeway 83
ICP 83, 95
protocol 83, 95
trace file example 218
TSI 46, 51
History of revisions 18

I

ICP
connect to 36
disconnect from 37
header 83, 95
reset/download 95
ICP buffers 41
ICP2432 embedded environment 30
Internet addresses 27
I/O
blocking vs non-blocking 32
completion handler (IOCH) 33, 143, 161, 162, 164, 165, 205
polling 207
signal processing 53
UNIX environment 205
VMS environment 209
VxWorks environment 208

L

LAN interface processor 22
Link
configure 36
connect to remote 36
data transfer 39
disconnect from remote 37
localAck DLI parameter 64, 134, 151, 164
Logging services 211
logLev DLI parameter 63, 64, 211
logLev TSI parameter 148, 149
logName DLI parameter 63, 211
logName TSI parameter 148

M

MaxBuffers TSI parameter 46, 51
maxBuffers TSI parameter 53, 79, 86, 148
MaxBufSize TSI parameter 42, 46, 47, 48, 49, 50, 51, 149
maxBufSize TSI parameter 53, 119, 125, 148, 215

maxConn TSI parameter 148
maxErrors DLI parameter 64, 81, 112, 126, 139, 193
maxInQ DLI parameter 53, 64, 81
maxOutQ DLI parameter 53, 65, 81
maxSess DLI parameter 53, 63, 79, 104, 107
Memory requirements 53
Message multiplexor 35
  connect to 36, 39
  disconnect from 37, 40
mode DLI parameter 65, 70, 82, 145, 158
Modes of operation
  normal and raw 84, 134
  raw 103, 123
msgBlkSize BSC parameter 41, 42
msgBlkSize DLI parameter 66
MuxCfg file 51
MuxCfg server configuration file
  Files
    server MuxCfg configuration 41, 50, 51

N

Negotiation of buffer size 49
Non-blocking I/O 32, 104, 108
  code example 167
  example program 157
  I/O completion handler 33
  signal processing 53
  TSI configuration 159
  UNIX 206
  VxWorks 208
Normal operation 34, 35
  see also Operation

O

On-line configuration file processing 61
Operating system
  Protogate's real-time 22
Operation
  normal 35
    configure link 36
    connect to ICP 36
    connect to MsgMux 36
    connect to remote 36
    connect to TSI 35
    disconnect from ICP 37
    disconnect from MsgMux 37
    disconnect from remote 37
    disconnect from TSI 38
    transfer data 37
  normal vs raw 34
  raw 38, 174
    connect to MsgMux 39
    connect to TSI 39
    data transfer 39
    disconnect from MsgMux 40
    disconnect from TSI 40
    example program 176
Optional arguments 38, 83, 174
  C data structure 83
Overview
  DLI 28
  DLI functions 74
  DLI hierarchy 35
  embedded ICP 22
  Freeway server 22
  product 21

P

PCIbus ICP2432 30
Polling I/O 207
portNo DLI parameter 65, 82
Product
  features 25
  overview 21
  support 19
Programs
  dlicfg preprocessor 32, 56, 99
  tsicfg preprocessor 32, 56
  tutorial examples 141
Protocol
  header 83, 95
  optional arguments 38, 83, 174
protocol DLI parameter 34, 38, 65, 70, 106, 145, 158

R

Raw operation 34, 38, 174
  example program 176
  see also Operation

Reference documents 14
Reset/download ICP 95
Resource requirements 53
reuseTrans DLI parameter 65, 190, 191
Revision history 18
rlogin 25

S

Server buffers 41
Server processor 22
server TSI parameter 149
Server-resident application 208
Session
  closing 28
  configuration 56, 61, 62
    DLI example 145, 158
    example 67
  main configuration 62
  memory requirements 53
  normal vs raw operation 34
  opening 28
  session ID 107
  sessionID 104
  status 80
Session status 106, 141, 182, 202
  DLI_STATUS_DEAD_SOCKET 202
sessPerConn DLI parameter 63
SIGALRM 206, 207
SIGIO 206
Signal processing 53
SNMP 25
Sockets
  dead socket detection 202
  dead status 202
Software
  download 27, 74, 75
  download using dlControl 95
Solaris 206
Status
  connection 49, 182
  session 106, 141, 182, 202
Support, product 19
Synchronous I/O
  see Blocking I/O
System
  configuration 47, 78
  resource requirements 53

T

TCP/IP 25
Technical support 19
telnet 25
timeout DLI parameter 103
timeout TSI parameter 53, 120, 126, 139, 149
traceLev DLI parameter 63, 65, 213
traceLev TSI parameter 148, 149
traceName DLI parameter 63, 116, 212
traceName TSI parameter 148
traceSize DLI parameter 63, 101, 116, 212
traceSize TSI parameter 148
Tracing services 211, 212
  binary format 221
  example output 218
  file layout 214
  Freeway server 222
transport DLI parameter 35, 39, 62, 65, 146
Transport subsystem interface
  see TSI
Transport subsystem interface (TSI) 27
transport TSI parameter 149
Troubleshooting 211
TSI 29, 35
  connect to 35, 39
  connection 38
    configuration 35, 39, 56, 62
    termination 37
  disconnect from 38, 40
  header 51
  see also Configuration, TSI
TSI buffer pool definition 46
tsicfg preprocessor program 32, 56
tsiCfgName DLI parameter 62, 63
Tutorial example programs 141

U

UNIX environment 205

V

VMS environment 209
VxWorks 22

VxWorks environment 208

W

WAN interface processor 22
wellKnownPort TSI parameter 149
writeType DLI parameter 34, 66, 134, 151, 163

Customer Report Form

We are constantly improving our products. If you have suggestions or problems you would like to report regarding the hardware, software or documentation, please complete this form and mail it to Protogate at 12225 World Trade Drive, Suite R, San Diego, CA 92128, or fax it to (877) 473-0190.

If you are reporting errors in the documentation, please enter the section and page number.

Your Name: ____

Company: ____

Address: ____

Phone Number: ____

Product: ____

Problem or Suggestion: ____

Protogate, Inc.
Customer Service
12225 World Trade Drive, Suite R
San Diego, CA 92128