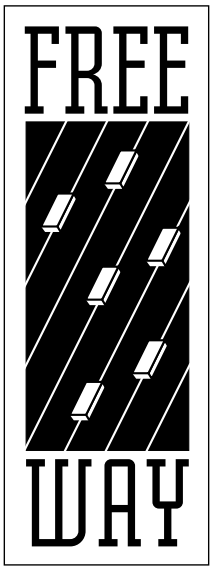
Preliminary
June 11, 2002

Freeway®
Protocol Software Toolkit
Programmer's Guide

DC 900-2007A



Protogate, Inc.
12225 World Trade Drive, Suite R
San Diego, CA 92128
November 1997

PROTOGATE

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

Freeway Protocol Software Toolkit Programmer's Guide

Printed in the United States of America

Contents

List of Figures

List of Tables

Preface

Purpose of Document

This document describes the protocol software toolkit for the Freeway server and Freeway embedded environments and the issues involved in developing software that executes on Freeway. It also provides information on client application programs and the host/ICP interface.

Intended Audience

This document should be read by programmers who are developing code to be downloaded to the ICP2432B. You should be familiar with your client system's operating system and with program development in a real-time environment. Familiarity with the C programming language and Motorola ColdFire® assembly language is helpful.

Required Equipment

You must have the following equipment to use the protocol software toolkit to develop and test communications applications:

- An ICP2432B installed in the Freeway server's backplane or embedded in your host computer system
- A console cable and an ASCII terminal or terminal emulator (running at 9600 b/s) for access to the ICP console port
- A set of Wind River tools for the Motorola ColdFire® processor

- If you plan to use the sample protocol software (SPS) test program as a basis for your client application code, you will need a C compiler for your client system

Organization of Document

Chapter 1 is an overview of the Freeway server and embedded products and the protocol software toolkit.

Chapter 2 describes the issues involved in ICP Wind River, including software-development tools, the various interfaces, and how to program the hardware devices.

Chapter 3 describes local memory address allocation on the ICPs.

Chapter 4 describes system download, configuration, and initialization.

Chapter 5 describes the ICP debugging tools and techniques.

Chapter 6 describes the ICP software.

Chapter 7 gives an overview of the host/ICP interface and describes the interface between the ICP's driver, XIO, and OS/Protogate application tasks.

Chapter 8 describes client applications.

Chapter 9 describes the messages exchanged between the client and the ICP.

Appendix A clarifies some points made in the technical manuals and describes some peculiarities of the devices and the ICP6000 hardware.

Appendix B provides some commonly used data rate time constants for SCC programming on the ICP6000.

Appendix C describes error codes.

Appendix D describes loopback test programs and sample programs.

References

Freeway general support:

- *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 Programmable Communications Servers Technical Overview* 25-000-0374
- *Freeway Software Release Addendum: Client Platforms* DC 900-1555
- *Freeway Embedded ICP2432 User's Guide for Windows NT* DC 900-1510
- *Freeway Server User's Guide* DC 900-1333

Freeway programming support:

- *Freeway Client-Server Interface Control Document* DC 900-1303
- *Freeway Data Link Interface Reference Guide* DC 900-1385
- *Freeway OS/Protogate Programmer's Guide* DC 900-2008
- *Freeway QIO/SQIO API Reference Guide* DC 900-1355
- *Freeway Server Software Toolkit Programmer's Guide* DC 900-1325
- *Freeway Transport Subsystem Interface Reference Guide* DC 900-1386
- *ICP2432B Hardware Description and Theory of Operation* DC 900-2006

Freeway protocol support:

- *Freeway ADCCP NRM Programmer's Guide* DC 900-1317
- *Freeway Asynchronous Wire Service (AWS) Programmer's Guide* DC 900-1324
- *Freeway BSC Programmer's Guide* DC 900-1340
- *Freeway BSCDEMO User's Guide* DC 900-1349
- *Freeway BSCTRAN Programmer's Guide* DC 900-1406
- *Freeway DDCMP Programmer's Guide* DC 900-1343
- *Freeway FMP Programmer's Guide* DC 900-1339
- *Freeway SIO STD-1200A (Rev. 1) Programmer's Guide* DC 900-1359

• *Freeway SWIFT and CHIPS Programmer's Guide*	DC 900-1344
• *Freeway Tactical Military Protocols Programmer's Guide*	DC 900-1341
• *Freeway X.25 Call Service API Guide*	DC 900-1392
• *Freeway X.25/HDLC Configuration Guide*	DC 900-1345
• *Freeway X.25 Low-Level Interface*	DC 900-1307

Other Documents (Available from Vendor):	**Vendor**
• *Z16C32 IUSC Integrated Universal Serial Controller* • *Technical Manual*	Zilog, DC8292-01

Other Documents (Development Tools and Environment):	**Vendor**
• *SingleStep Debugger for the ColdFire® Microprocessor Family*	WRS

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 3100, Freeway 3200, Freeway 3400, or Freeway 3600), or to the Freeway embedded product (for example, the embedded ICP2432B).

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 sample applications have the format: `sps`*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 ® Protocol Software Toolkit Programmer's Guide*, Protogate document DC 900-2007A, is recorded below:

Document Revision	Release Date	Description
DC 900-1338A	November 4, 1994	Original release
DC 900-1338B	November 22, 1994	Update file names for Release 2.1 Add Appendix D, "Test Programs"
DC 900-1338C	July 1995	Update file names Add ICP2424 information
DC 900-1338D	February 1996	Minor modifications throughout Add ICP6030 information Add new `dlControl` function to Table 8–3 on page 125 Add Windows NT to Appendix D Delete HIO task information
DC 900-1338E	November 1997	Add Freeway embedded product information Add ICP2432 information Document changes in directory structure
DC 900-2007A	June 2002	Port the 1338 document to Protogate, Inc.

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 Introduction

1.1 Freeway Overview

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

Both the Freeway Server product and the Freeway Embedded product use the same Freeway application program interface (API). Therefore, migration between the two Freeway environments simply requires linking your client application with the proper library. Freeway supports various client operating systems (for example, UNIX, VMS, and Windows NT).

Protogate protocols that run on the Freeway intelligent communications processors (ICPs) are independent of the client operating system and the hardware platform (Freeway Server or Freeway Embedded).

1.1.1 Freeway Server Product

Protogate's Freeway communications servers enable client applications on a local-area network (LAN) to access specialized WANs through the Freeway API. 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 Server product configuration.

To maintain high data throughput, the Freeway Server 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 operating system to provide a full-featured base for the LAN interface and layered services needed by Freeway.

The Freeway Server 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.

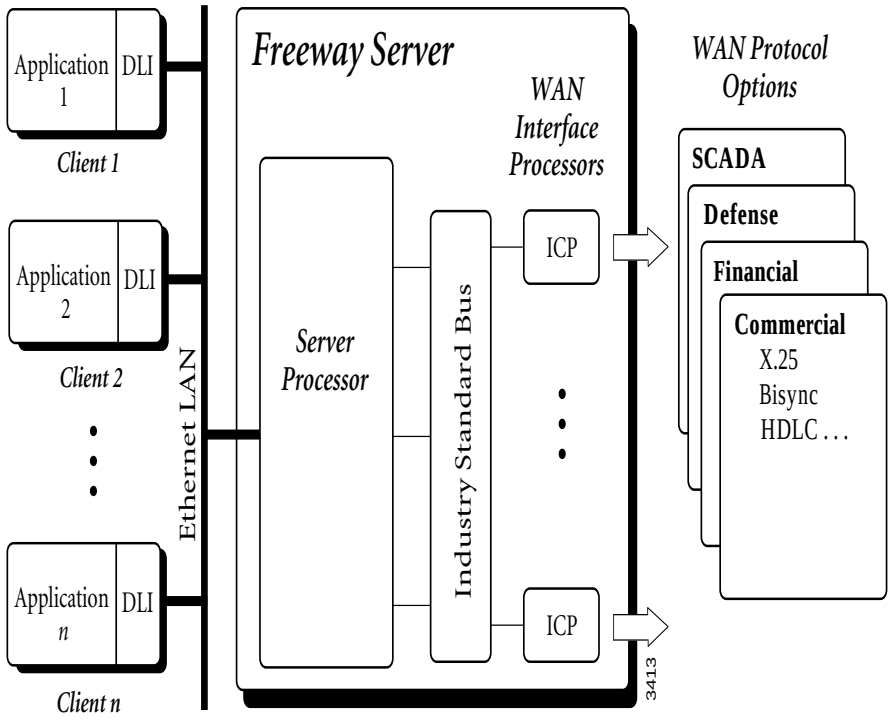


Figure 1–1: Freeway Server Product Configuration

1.1.2 Freeway Embedded Product

The Freeway Embedded product connects your computer directly to the WAN (for example, through Protogate's Embedded ICP2432B PCIbus board). The Freeway Embedded product provides client applications with the same WAN connectivity as the Freeway Server product, using the same Freeway API. The ICP runs the communication protocol software using Protogate's real-time operating system. Figure 1–2 shows the Freeway Embedded product configuration.

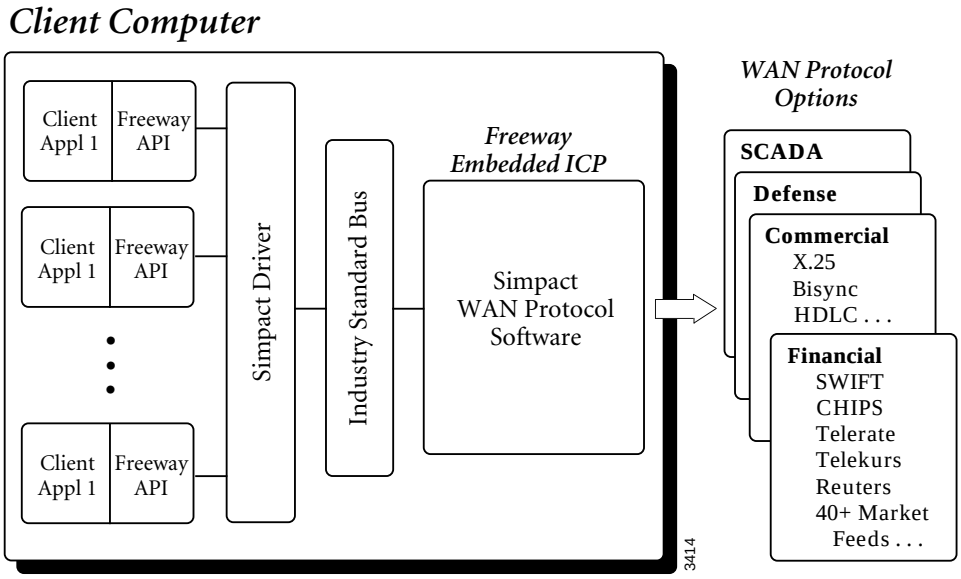


Figure 1–2: Freeway Embedded Product Configuration

Summary of Freeway features:

- Support for multiple ICPs (two, four, or eight communication lines per ICP)
- Wide selection of electrical interfaces including EIA-232, EIA-449, EIA-530, EIA-562, V.35, ISO-4903 (V.11), and MIL-188
- Variety of off-the-shelf communication protocols available from Protogate which are independent of the client operating system (for example, Windows NT, UNIX, or VMS) and hardware platform (Freeway Server or Freeway Embedded)
- Support for multiple WAN communication protocols simultaneously
- Elimination of difficult LAN and WAN programming and systems integration by providing a powerful and consistent Freeway application program interface (API)
- Creation of customized server-resident and ICP-resident software, using Protogate's Wind River toolkits
- Freeway Server standard support for 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 Environments

1.2.1 Freeway Server 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 the Freeway server, 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 server's `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 server connected to a locally attached client by a TCP/IP network across an Ethernet LAN interface. Running a client application against a Freeway server in a client-server environment requires the basic steps described in Section 1.2.1.1, Section 1.2.3, and Section 1.2.4.

1.2.1.1 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 Server User's Guide* to initially set up your Freeway server and download its operating system, server, and protocol software.

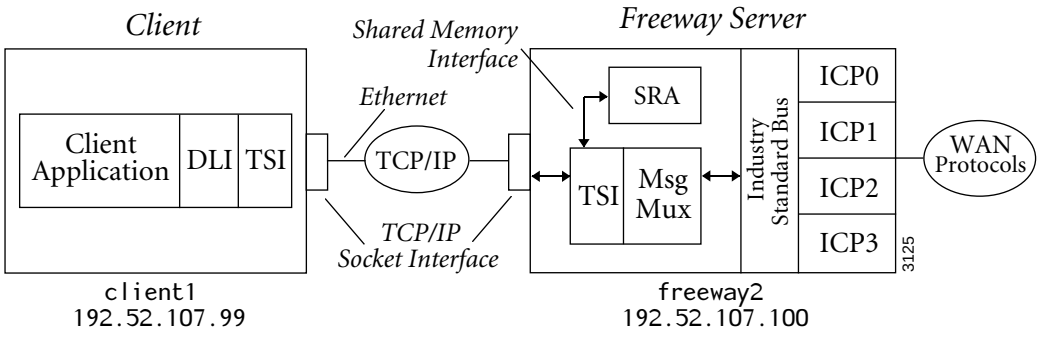


Figure 1–3: A Typical Freeway Server Environment

1.2.2 Freeway Embedded Client-Service Environment

In the Freeway embedded environment, the client application still interfaces with the Freeway APIs, DLI and TSI. The primary difference is that the TSI layer now communicates with the ICP driver instead of with TCP/IP and `msgmux`.

Figure 1–4 shows a typical Freeway embedded environment. Running a client application requires the basic steps described in Section 1.2.3 and Section 1.2.4.

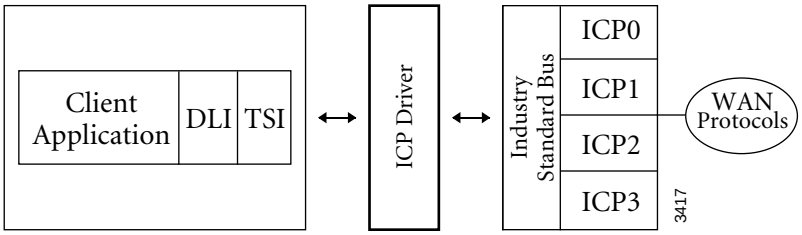


Figure 1–4: A Typical Freeway Embedded Environment

1.2.3 Defining the DLI and TSI Configuration

In order for your client application to communicate with the ICP's protocol task, you must define the DLI sessions and the transport subsystem interface (TSI) connections. You have the option of also defining the protocol-specific ICP link parameters. To accomplish this, you first define the configuration parameters in DLI and TSI ASCII configuration files, then you run two preprocessor programs, `dlicfg` and `tsicfg`, to create binary configuration files. The `dlInit` function uses the binary configuration files to initialize the DLI environment.

1.2.4 Application Operations

1.2.4.1 Opening a Freeway Session

After the DLI and TSI configurations are properly defined, your client application program uses the `dlOpen` function to establish a DLI session with an ICP link. For the Freeway server, the DLI establishes a TSI connection with `msgmux` through the TCP/IP BSD-style socket interface as part of the session establishment process. For Freeway embedded systems, the DLI establishes a TSI connection directly to the ICP driver.

1.2.4.2 Exchanging Data with the Remote Application

After the link is enabled, the client application program can exchange data with the remote application using the `dlRead` and `dlWrite` functions.

1.2.4.3 Closing a Freeway 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 the Freeway server or ICP driver.

1.3 Protocol Toolkit Overview

The protocol software toolkit helps you develop serial protocol applications for execution on Protogate's intelligent communications processors. Many of the software modules required to build a complete system are provided with the toolkit or reside in the ICP's PROM, including download facilities, operating system, and the Peeker (ICP2432B) debugging tool. The toolkit also includes a debug monitor program for use with Wind River Systems' SingleStep debugger. (The SingleStep debugger must be purchased directly from Wind River Systems.) All you have to provide is your application code, which you can build using the toolkit's sample protocol software as a model. Chapter 2, Chapter 4, and Chapter 5 give more information on Wind River, configuration, and debugging.

The toolkit includes software, provided on the distribution media, and complete documentation (see the document "References" section in the *Preface*). Some of the toolkit's software components, such as the SingleStep monitor, are provided only in executable object format. All other components are provided in both source and executable form so that they can be modified, used as coding examples, or linked with user applications. Figure 1–5 shows a block diagram of the ICP's PROM and the toolkit's software components for the Freeway server. Figure 1–6 shows the same information for the Freeway embedded products.

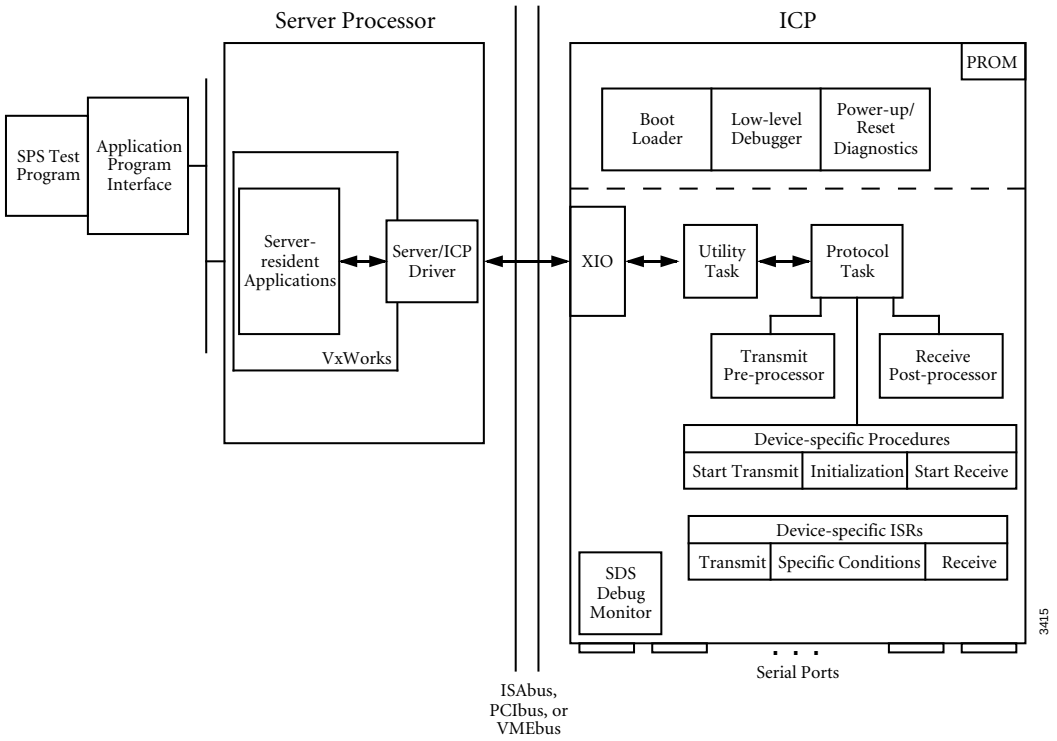


Figure 1–5: ICP PROM and Toolkit Software Components - Freeway Server

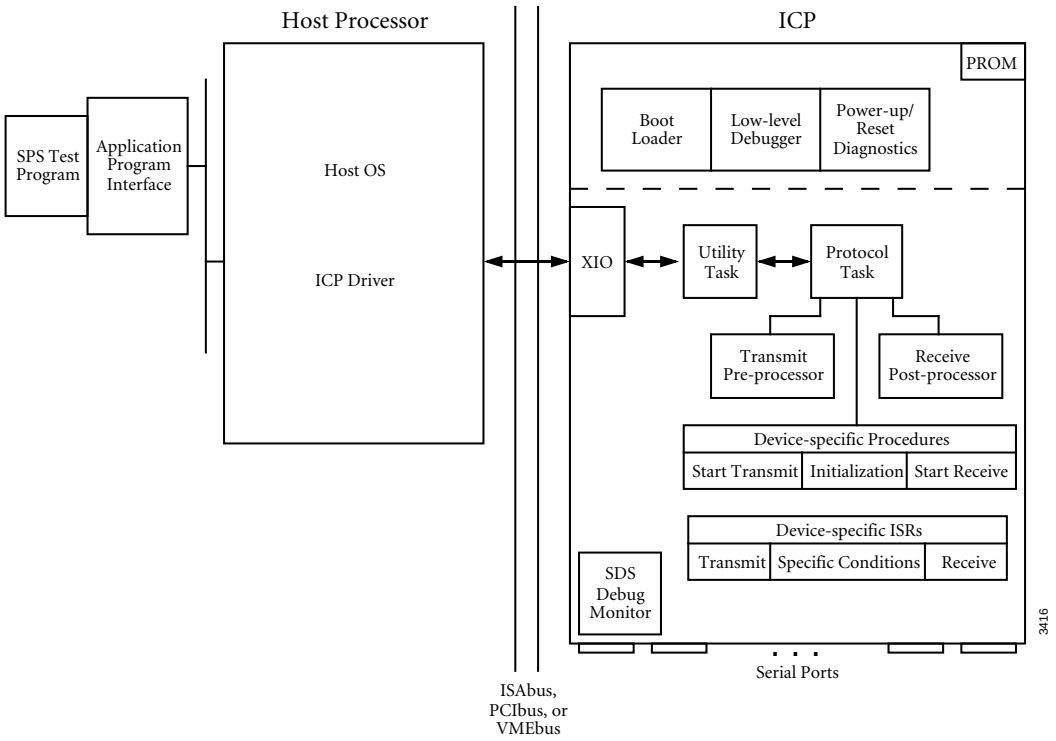


Figure 1–6: ICP PROM and Toolkit Software Components - Freeway Embedded

1.3.1 Toolkit Software Components

The toolkit loopback test programs (`spsalp.c` and `spsslp.c`) are provided in source form and, when compiled, execute in the client application program's system environment. For the test procedures, see Appendix D in this manual and the "Protocol Toolkit Test Procedure" appendix in the *Freeway Server User's Guide* or the appropriate Freeway embedded user's guide.

The following programs execute on the ICP:

- System-services module containing the OS/Protogate operating system and the XIO ICP-side driver (sources provided)
- Sample protocol software (source provided)
- Sample host interface I/O utility (source provided)
- Debug monitor; must be used with the Wind River Systems' SingleStep monitor package (executable code only)

The following source files aid in ICP Wind River:

- Subroutine library for C interface to OS/Protogate
- Macro library for assembly interface to OS/Protogate
- Header files with OS/Protogate and XIO definitions and equates
- Make files for supplied source files
- `*.dld` files for linking and address resolution of the executable images

Chapter

2 Wind River for the ICP

This chapter describes the issues involved in developing software for the Protogate ICPs, including software-development tools, the client application program interfaces, and the hardware devices. The application program interface between the client and ICP protocol tasks are described in the *Freeway Transport Subsystem Interface Reference Guide* and *Freeway Data Link Interface Reference Guide*. The interface between the ICP and the server (for Freeway server systems) or remote (for Freeway embedded systems) is described in Chapter 7 of this manual.

2.1 Board-level Protocol-executable Modules

An ICP board-level protocol-executable module is an absolute image file containing Motorola ColdFire® code and data developed on a Diab development system and subsequently downloaded to the ICP. Any division of code and data among modules is entirely arbitrary. For example, Protogate's protocol software toolkit includes the following modules:

- A system-services module containing the OS/Protogate operating system kernel, timer task, and XIO for the ICP2432B (`osp_2432B.mem`)
- A module comprising the sample protocol application for the ICP2432B (`sps_fw_2432B.mem`)
- A module containing the source-level debug monitor for the ICP2432B (`icp2432Bc.mem`) used only with the Wind River Systems' SingleStep debugger

In general, the toolkit programmer develops or modifies one or more application modules or tasks that run with Protogate's system-services module. Application tasks can run concurrently.

Modules are downloaded to the ICP as individual entities as described in Chapter 4. They are not linked with one another. Any shared information must be made available to a module when it is created (in other words, during compilation or assembly) or must be obtained by the module at the time of execution. Modules designed to execute in the OS/Protogate environment access system services through the use of software traps and, in general, communicate with other tasks through OS/Protogate services, using public task and queue IDs.

For these reasons, and because there are no provisions in the OS/Protogate environment for memory protection, it is essential to document the system resources required by a module if it is to execute in combination with other modules. The following information is provided for each module developed by Protogate and defined in the `*.spc` files:

- Reserved areas of memory for code, data, and stack space
- Reserved exception vector table entries
- Dependencies on, or conflicts with, other modules
- Configuration requirements (number of tasks, priorities, queues, alarms, resources, and partitions for the configuration table parameter list)
- Task initialization structures to be included in the configuration table
- Reserved task, queue, alarm, resource, and partition IDs (to avoid conflict with user-added modules and as public information for intertask communication)

During the design and development of your application, you can use this information to build a complete system composed of compatible and cooperating modules. In addi-

tion, your application code must provide a system configuration that is adequate for the combined needs of all the modules in the system, and it must include the required task initialization structures.

2.2 Development Tools

Modules are developed at Protogate using Wind River Systems' Diab C/C++ cross-compiler, assembler, and linker, and the SingleStep debugger. This section describes the issues related to the development of download modules from the perspective of the tools that Protogate has chosen.

2.2.1 WRS Compiler/Assembler/Linker

Protogate has worked with Wind River Systems (WRS) to offer source-level debugging for the toolkit using the WRS SingleStep debugger for the ColdFire® microprocessor family. To use the SingleStep debugger, see Chapter 5.

The WRS tools are available on SUN UNIX workstations and PCs running Windows. The Diab C/C++ cross-compiler and SingleStep debugger must be purchased directly from Wind River Systems.

The following WRS documents apply to these development tools:

- *Diab C/C++ for the ColdFire® Microprocessor Family*
- *SingleStep Debugger for the ColdFire® Microprocessor Family*

The Diab C/C++ package is designed specifically for the Motorola ColdFire® family and includes a complete development system with a C compiler, an assembler, a linker, and a downloader. The WRS assembler allows you to define multiple relocatable regions, identified by region names. These regions are mapped into the target memory structure by the linker using a linker specification file. This file allows you to map various regions to particular addresses and position them in ROM or RAM as needed. The C compiler automatically splits output into five standard regions for code, strings, con-

stant data, initialized data, and uninitialized data. The `freeway/icpcode/proto_kit/icp2432B`[1] directory contains a sample make file (`makefile`) and two sample linker specification files (`sps.dld`, `sps_os.spc`) which can be used to build the `sps_fw_2432B.mem` image. The second file is provided so that the source for OS/Protogate will also be accessible during debugging.

2.3 Interfacing to the Operating System

The assembly and C language interfaces to OS/Protogate are described in the *Freeway OS/ProtogateProgrammer's Guide*. The `freeway/icpcode/proto_kit/src` directory contains source code for a C interface library (`oscif.h` and `oscif.asm`). The routines in this library are written according to the subroutine calling conventions of the Diab compiler and can be easily modified for most other C compilers or high-level language compilers.

The interface routines are necessary when accessing OS/Protogate from C language routines for two reasons. First, OS/Protogate's system calls are accessed through a software trap instruction, which cannot be generated directly from C. Second, the subroutine calling conventions of the Diab compiler (where parameters are passed mainly on the stack) differ from those of the OS/Protogate system calls (where parameters are passed in registers). The interface routines must perform the necessary translations before and after OS/Protogate system calls.

The `oscif.h` file contains C structure definitions for all relevant operating system data structures.

For programs written in assembly language, the `freeway/icpcode/proto_kit/src` directory includes the files `sysequ.asm`, with OS/Protogate system call macros, and `oscif.asm`, with assembly language definitions of OS/Protogate data structures. These

1. `icp`*nnnn* refers to the `icp2424`, `icp2432`, `icp6000`, or `icp6030` directory.

files are in a format compatible with the Diab assembler, but can also be modified for use by other assemblers.

2.4 Motorola ColdFire® Programming Environment

The Motorola ColdFire® CPU is a 32-bit microprocessor with 32-bit registers, internal data paths, and addresses that provides a four-gigabyte direct addressing range. If your application code will be written in assembly language, you will find the *ColdFIre® Microprocessor Family Programmer's Reference Manual* (Motorola) indispensable. It contains information on the general-purpose and special registers, addressing modes, instruction set, and exception processing. When programming in a higher-level language, most aspects of the processor are relatively transparent. The following sections present some general information to help you understand the ColdFire® programming environment.

2.4.1 Processor Privilege States

The ColdFire® supports two privilege levels: user and supervisor. On the ICP, OS/Protogate operates in supervisor state, as do all interrupt service routines and certain sections of the application code. All tasks (including the system-level timer) operate in user state, where certain operations are not allowed. See the *ColdFIre® Microprocessor Family Programmer's Reference Manual* (Motorola) for additional information.

2.4.2 Stack Pointers

The ColdFire® special registers include a system stack pointer (SSP).

A stack pointer is pre-decremented when an element is added to the stack (pushed) and post-incremented when an element is removed (popped). Stacks therefore grow from higher to lower memory addresses, and the stack pointer always contains the address of the element currently at the top of the stack.

During its initialization, OS/Protogate allocates space for the system stack and initializes the SSP. The system stack is used whenever the processor is in supervisor state. This includes system calls and all interrupt service routines, including those associated with user applications.

You must allocate stack space for each application task you create and specify the initial stack pointer in the task initialization structure (see Section 4.2.2 on page 53). The initial stack pointer should be specified as the ending address of the stack space plus one. For example, if a task's stack space is 0x40116000 through 0x410163FF, the initial stack pointer should be specified as 0x40116400. OS/Protogate saves this initial value in the task control block as the current stack pointer. When the task is dispatched, OS/Protogate initializes the SP to the stack address saved in the task control block. When the task is preempted, the task's state (the contents of the general registers) is saved on its stack and the current SP is again saved in the task control block.

When allocating a task's stack, you must consider the space required at the deepest level of nested subroutine calls, and allow 66 bytes for the registers saved when the task is preempted. You need to allocate additional stack space for interrupt service routines, as the SSP is used for interrupt processing.

Note

The stack spaces are defined in the linker specification file `freeway/icpcode/proto_kit/icp2432B/sps.dld`.

2.4.3 Exception Vector Table

On the ColdFire®, interrupts and traps are processed through an exception vector table. The ColdFire® vector base register points to the exception vector table, which contains 256 longword (four-byte) vectors. The vector base register is not accessible in user state, so OS/Protogate provides the base address of the exception vector table in its system address table. (See the *Freeway OS/ProtogateProgrammer's Guide.*)

The *ColdFIre® Microprocessor Family Programmer's Reference Manual* (Motorola) lists vector assignments as defined by the ColdFire® CPU. Table 2–1 lists the vectors that are reserved for use by Protogate's system software.

Table 2–1: Vectors Reserved for System Software

Vector Number (Decimal)	Vector Offset (Hexadecimal)	Function
25	64	Auto vector level 1
26	68	Auto vector level 2
27	6C	Auto vector level 3
28	70	Auto vector level 4
32	80	TRAP # 0
33	84	TRAP # 1
34	88	TRAP # 2
35	8C	TRAP # 3
36	90	TRAP # 4
37	94	TRAP # 5
47	BC	TRAP # 15

To install an interrupt service routine (ISR) for a particular device, multiply the vector number by four to obtain the vector offset, add the offset to the base address of the exception vector table, and store your ISR entry point at the resulting address.

When the device generates an interrupt, it supplies the ColdFire® CPU with the eight-bit vector number, which the CPU multiplies by four to obtain a vector offset, then adds the contents of the vector base register to obtain the vector address at which your ISR entry point is stored. When interrupt servicing is complete, the ISR must terminate with a "return from ISR" (`s_iret`) system call (described in the *Freeway OS/ProtogateProgrammer's Guide*) if the interrupt requires that system services be invoked. Otherwise, a return from exception (RTE) is sufficient.

When programming interrupt service routines in a high-level language, it is usually necessary to provide an assembly language "shell" for the ISR in order to save certain registers.

For example, the Diab compiler saves on entry and restores on exit all registers used in a subroutine except D0, D1, A0, and A1, which are considered working registers. The calling code must save these registers, if necessary, before making a subroutine call. These calling conventions, however, are not sufficient for ISRs. An ISR is not "called" in the ordinary sense; it interrupts code that might currently be using the working registers. The ISR must, therefore, save those registers as well.

Because many compilers cannot distinguish between an ordinary subroutine and an interrupt service routine, the programmer must provide an assembly language shell to save the working registers on entry and restore them at completion of the ISR. (Note that it is the address of the shell rather than the high-level language routine that must be stored in the appropriate vector of the exception vector table.) Figure 2–1 shows a sample assembly language shell.

```
   SECTION          9
   XREF       _Cisr                 external reference to C isr
   XDEF       _isr_shell            external definition for C code
*                                   which stores this address
*                                   in the exception vector table
_isr_shell
   movem.l    d0/d1/a0/a1,-(sp)     save registers not saved by C
   jsr        _Cisr                 call C routine for interrupt
*                                   processing
   movem.l    (sp)+,d0/d1/a0/a1     restore registers
   s_iret                           return from isr (system call)
```

Figure 2–1: Assembly Language Shell

2.4.4 Interrupt Priority Levels

The Motorola ColdFire® supports seven levels of prioritized interrupts, with level 7 being the highest priority. Any number of devices can be chained to interrupt at the

same priority. Table 2–2 shows the interrupt priorities for the various ICP's hardware devices.

Table 2–2: ICP Interrupt Priority Assignments

Device(s)	Level
ICP2432B	
NMI and Bus Error Logic	7
Integrated Universal Serial Controllers (IUSC)	5
Integrated periodic timer interrupt	4
PCIbus	1

When an interrupt occurs at a particular priority, the interrupt mask field in the ColdFire®'s status register is set to the priority level of that interrupt, causing other interrupts at the same or lower priorities to be ignored. When interrupt servicing is complete, the interrupt mask level in the status register is returned to its previous value, at which time pending interrupts at lower priorities can be serviced.

The interrupt priority level can be changed by directly modifying the mask field in the status register, but this is possible only in supervisor state. OS/Protogate includes a system call that can be called from the task level to modify the interrupt priority level.

The *ColdFIre® Microprocessor Family Programmer's Reference Manual* (Motorola) contains important information that should be studied before implementing interrupt-level code.

2.5 ICP2432B Hardware Device Programming

The ICP2432 uses the Motorola ColdFire® CPU. The ColdFire® includes:

- Integer Arithmetic CPU

- a two-channel DMA controller
- a two-channel universal asynchronous receiver/transmitter (UART)
- a periodic interrupt timer
- two counter/timers
- A parallel digital port used as an LED register

In addition to the Motorola ColdFire®, the ICP2432B's programmable devices include:

- two, four, or eight Z16C32 integrated universal serial controllers (IUSCs) with integral DMA
- Sipex's SP503 (ICP2432B-4) or SP506 (ICP2432B-2) multi-mode serial transceivers
- a test mode register

Note

The 8-port ICP2432B only supports EIA-232.

2.5.1 Programming the ColdFire®

The ColdFire®'s serial port 1 is used as a console port. The second serial port is earmarked for use as a printer port or for use by the SingleStep debugger.

The ColdFire®'s parallel port control the red and green LEDs on the mounting bracket. Table 2–3 contains the information needed to turn the green and red LEDs on and off.

OS/Protogate uses the periodic interrupt timer, which uses vector 0x40.

2.5.2 Programming the Integrated Universal Serial Controllers

The Z16C32 IUSCs are used to control the ICP's serial ports. Each IUSC controls transmit and receive operations for one port. The IUSC also includes a DMA facility. Refer

Table 2–3: LED Control Information

Address	Value	Operation
0x3000_0248	0x01	Green LED on
0x3000_0248	0x02	Red LED on
0x3000_0248	0x00	Both on
0x3000_0248	0x03	Both off

to the *Z16C32 IUSC Integrated Universal Serial Controller Technical Manual*, for IUSC programming instructions. The sample protocol software package includes examples of IUSC programming for asynchronous, byte synchronous and bit synchronous communications. See Chapter 6 for more information.

2.5.3 Programming Sipex's Multi-Mode Serial Transceivers

The ICP2432B uses the SP503 or SP506 multi-mode serial transceivers. These transceivers allow software to select the electrical protocol to be used while communicating on the serial line. Table 2–4 gives the value to be written into the transceiver to select the corresponding electrical interface. See Chapter 3 for the addresses of the transceivers.

Table 2–4: SP503 or SP506 Electrical Interface Values

Interface	Value
RS-232	0x02
RS-422 w/0 term	0x04
RS-422 w term	0x05
RS-449 or EIA-530	0x0d
V.35	0x0e

2.5.4 Programming the Test Mode Register

All modem control signals except Test Mode are handled directly by the IUSC associated with the port. The Test Mode input status for all supported ports is through the Test Mode register located at 0x6001_0000. When a bit is set to one, the Test Mode signal is asserted on the serial line. See Figure 2–2.

Address = 0x60010000, byte wide, read only

7	6	5	4	3	2	1	0

Figure 2–2: Test Mode Register, ICP2432

Chapter

3 Memory Organization

This chapter describes the memory maps for the ICP2432B.

3.1 ICP2432B

The 64-kilobyte EPROM (Flash) on the ICP2432B is located at address 0x0000_0000. The EPROM contains the diagnostics, PEEKER debugging tool, and boot loader.

Thirty-two (32) megabytes of synchronous dynamic random access memory (SDRAM) starts at 0x4000_0000. Memory addresses 0x4000_0000 to 0x4010_0000 are reserved. The system services module (containing the operating systems and XIO) is loaded beginning at address 0x4010_0000. As described in Section 4.3.1 on page 56, the fixed memory requirements for a particular version of the system services module are specified in the `spsdefs.h` file, and additional memory required for the OS/Protogate's configurable data section depends on the system configuration. The rest of the SDRAM is available for user applications.

The *ICP2432B Hardware Description and Theory of Operation* provides a complete memory map. Table 3–1 summarizes the hardware device and register addresses.

Table 3–1: ICP2432B Device and Register Addresses

Device or Register	Base Address (Hexadecimal)
Base address of IUSC for Port 0	40000_0000
Base address of IUSC for Port 1	40000_1000
Base address of IUSC for Port 2	40000_2000
Base address of IUSC for Port 3	40000_3000
Base address of IUSC for Port 4	40000_4000
Base address of IUSC for Port 5	40000_5000
Base address of IUSC for Port 6	40000_6000
Base address of IUSC for Port 7	40000_7000
SP503 or 506 for Port 0	40000_8000
SP503 or 506 for Port 1	40000_9000
SP503 or 506 for Port 2	40000_A000
SP503 or 506 for Port 3	40000_B000

Chapter

4 ICP Download, Configuration, and Initialization

Section 4.1 of this chapter describes additional download considerations not covered in the *Freeway Server User's Guide* or the Freeway embedded user's guide so you can download the toolkit protocol software with or without the Wind River Systems (WRS) debug monitor. Section 4.2 describes configuration and initialization issues. Section 4.3 describes the relationship between the system configuration and OS/Protogate's memory requirements and performance.

4.1 Download Procedures

4.1.1 Freeway Server Download Procedure

The protocol software toolkit installation procedure is described in the *Freeway Server User's Guide*. On UNIX systems, all subdirectories are installed by default under the directory named `/usr/local/freeway`. On VMS systems, all subdirectories are installed by default under the directory named `SYS$SYSDEVICE:[FREEWAY]`. On Windows NT systems, all subdirectories are installed by default under the directory named `c:\freeway`. ***It is highly recommended that you use these default directories.***

During the software installation, boot, and test procedures described in the *Freeway Server User's Guide*, the non-debug version of the toolkit software is downloaded to the ICP. However, during toolkit application development, you must modify your Freeway server boot configuration file and then reboot the Freeway server to download and start the debug monitor module. Section 4.1.1.1 and Section 4.1.1.2 describe the files and modifications required to download with or without the Wind River Systems (WRS) SingleStep monitor.

The Freeway server boot configuration file, used to control the download procedure, is covered in detail in the *Freeway Server User's Guide*. The boot configuration file is located in the `freeway/boot` directory (for example, `bootcfg.pci` for a Freeway 3100/3200/3400/3600). The download script file parameter (`download_script`) in the boot configuration file specifies the modules to be downloaded to the ICP and the memory location for each module. You must modify the `download_script` parameter as described in Section 4.1.1.2 when you need to change between debug and non-debug operation.

When you reboot the Freeway server, the modules are downloaded to the ICP in two stages. First, the server software uses a file transfer program to download the modules to the server's local memory. The modules are then transferred across the PCIbus to the ICP.

PCIbus transfers are handled by the ICP's CPU. The server software provides the location and size of the binary images and the address in the ICP's SDRAM at which the modules should be loaded, and then signals the ICP to begin the download process.

4.1.1.1 Downloading Without the Debug Monitor

Under normal operations you download the toolkit software without the debug monitor. The following files are required:

spsload — This is the download script file. You must specify this file name for the `download_script` parameter in your boot configuration file. The file is in the `freeway/boot` directory.

osp_2432B.mem — This is the system-services module containing the OS/Protogate operating system kernel, timer task, and XIO. This file is in the `freeway/boot` directory.

sps_fw_2432B.mem — This is the toolkit sample protocol software (SPS) module. This file is in the `freeway/boot` directory. Source files are in the `freeway/icpcode/proto_kit/src` directory. If you make changes to the source files, you must rebuild the `sps_fw_2432b.mem` module before downloading. The makefile is in the `freeway/icpcode/proto_kit/icp2432b` directory.

Figure 4–1 shows the `spsload` download script file that downloads the toolkit software when you reboot the Freeway server. Uncomment the "normal" lines associated with the type of ICP you are using and modify path names as needed. Do not change the memory locations (such as `40100000`) for the `LOAD` commands.

```
# Protocol load files are referenced from the server boot configuration file
#
# load files contain LOAD and INIT commands.
#    LOAD <fully qualified path name to the .mem file> <load address>
#    INIT <initialization address>
#
# each protocol toolkit load file must contain an osimpact .mem file,
# a protocol toolkit .mem file and a buffer size file.
#
# Uncomment the ICP load/init section below for your ICP model and
# modify the path to match the actual installation path. The examples
# below are for the default UNIX installation. (see the bootcfg example
# file for example path syntax for various host machines)
#
# the below is an example for the icp2432B normal
#
#LOAD /usr/local/freeway/boot/osp_2432B.mem             40100000
#LOAD /usr/local/freeway/boot/snmp_2432B.mem            40110000
#LOAD /usr/local/freeway/boot/sps_2432B.mem             40120000
#LOAD /usr/local/freeway/boot/buffer.size               4011fff0
#INIT                                                   40120000
#
# the below is an example for the icp243B2 debug
#
#LOAD /usr/local/freeway/boot/osp_2432B.mem             40100000
#LOAD /usr/local/freeway/boot/snmp_2432B.mem            40110000
#LOAD /usr/local/freeway/icpcode/boot/icp2432bc.mem     40001000
#LOAD /usr/local/freeway/boot/sps_2432B.mem             40120000
#LOAD /usr/local/freeway/boot/buffer.size               4011fff0
#INIT                                                   40001000
```

Figure 4–1: Protocol Toolkit Download Script File (spsload)

4.1.1.2 Downloading With the SingleStep Monitor

During application development you must download the toolkit software with the debug monitor. The WRS tools are not compatible with VMS platforms, but Windows versions are available. If you are a VMS user, you can develop and debug your software with these tools using a PC running under Windows and a utility to transport files from the PC to the VMS system. Chapter 5 explains how to use the WRS debug tools.

The following files are required:

spsload — This is the download script file. You must specify this file name for the `download_script` parameter in your boot configuration file. The file is in the `freeway/boot` directory.

icp2432bc.mem — This module contains the source-level debug monitor. This file is in the `freeway/icpcode/proto_kit/icpboot` directory.

sps_fw_2432b.mem — This is the toolkit sample protocol software (SPS) module. This file is in the `freeway/boot` directory. Source files are in the `freeway/icpcode/proto_kit/src` directory. If you make changes to the source files, you must rebuild the `sps_fw_2432b.mem` module before downloading. The makefile is in the `freeway/icpcode/proto_kit/icp2432b` directory.

Figure 4–1 on page 48 shows the `spsload` download script file that downloads the toolkit software when you reboot the Freeway server. Uncomment the "debug" lines associated with the type of ICP you are using and modify path names as needed. Do not change the memory locations (such as `40100000`) for the `LOAD` commands.

When the WRS debug monitor is downloaded along with other executable image files, the placement and order of execution of the downloaded code is different. The download addresses of the modules can differ, and the debug module will be first to execute.

Note that the monitor must use SDRAM from 0x40110000 to 0x40120000 on the ICP2432B.

4.1.2 Freeway Embedded Download Procedure

As with the Freeway server environment described in Section 4.1.1, the `freeway/boot/spsload` file defines the files to be downloaded to the embedded ICP. Uncomment the lines associated with the type of ICP you are using and modify path names as needed. Do not change the memory locations (such as `40100000`) for the `LOAD` commands.

The ICPs are loaded by the program `icpload` (a Windows NT service) which is normally executed during the start up of the host system. During development, the ICPs may be loaded or reloaded by running `spsload`.

4.2 OS/Protogate Configuration and Initialization

A complete ICP run-time system is composed of a system-services module and one or more user-application modules. One of the user-application modules must include a configuration table and a system task initialization routine. For example, the system-services module provided with toolkit is the binary image file (`osp_2432B.mem`), and the sample user-application modules are the sample protocol software binary image (`sps_fw_2432B.mem`).

The last step of the download script file specifies an entry point or start-up address for execution of the downloaded code (see the `INIT` command in Figure 4–1). This entry point must be the address of your system task initialization routine (or the address of the `icp2432Bc.mem` debug module if you are running with the WRS debug monitor).

Figure 4–2 shows a sample memory layout that specifies the download and start-up locations in the ICP2432B's RAM for the system-services module and sample protocol application.

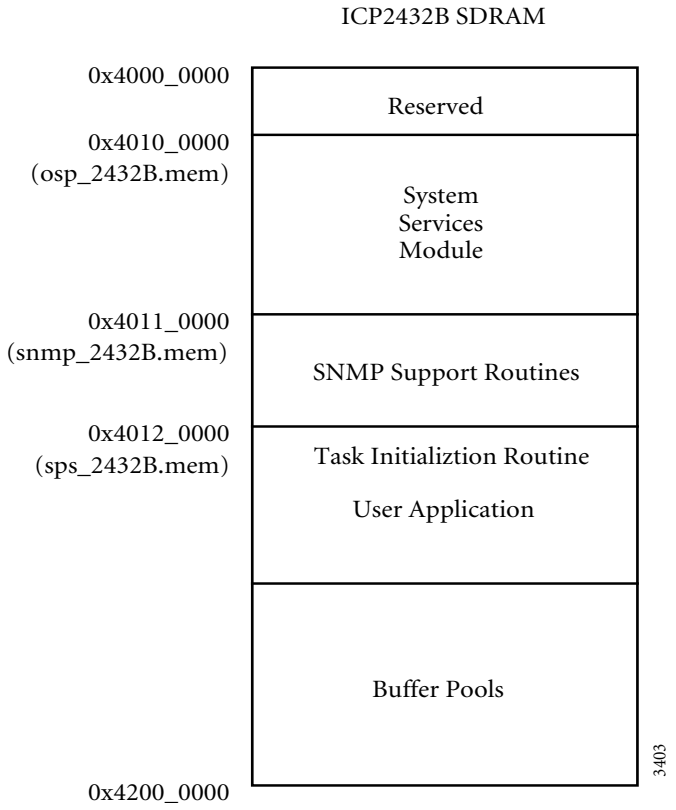


Figure 4–2: ICP2432B Memory Layout with Application Only

Figure 4–3 shows a similar ICP2432B configuration consisting of the system-services module, WRS debug monitor, and sample protocol application.

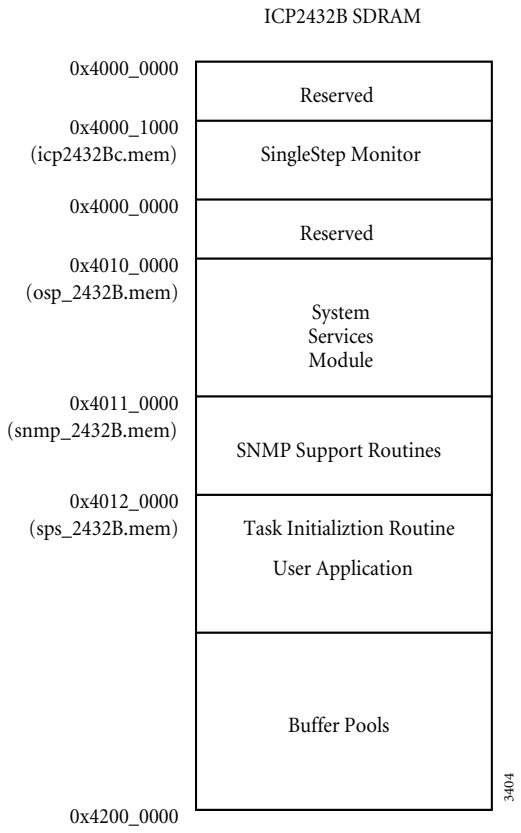


Figure 4–3: ICP2432B Memory Layout with Application and SingleStep Monitor

4.2.1 Configuration Table

The format of the configuration table is defined in the *Freeway OS/ProtogateProgrammer's Guide* and consists of a list of configurable parameters and a list of task initialization structures.

OS/Protogate creates its data structures based on the values of the parameters, then creates a task for each task initialization structure.

Section 4.3 discusses the selection of appropriate configuration parameters. Figure 4–4 gives an example of a configuration table (not including the task initialization structures).

```
_spsccf
    DC.W         5       number of tasks
    DC.W         4       number of priorities
    DC.W       350       number of queues
    DC.W        32       number of alarms
    DC.W         4       number of partitions
    DC.W         0       number of resources
    DC.W        10       tick length (milliseconds)
    DC.W         0       ticks for time slice
    DC.L         0       no user clock isr
```

Figure 4–4: Sample Configuration Table

4.2.2 Task Initialization Structures

A list of task initialization structures must follow the configuration table. The sample configuration table shown previously in Figure 4–4 is repeated in Figure 4–5 with task initialization structures for a sample task.

```
*
* Configuration Table
*
        .text

        .global _spscfg

_spscfg
        DC.W    5       number of tasks
        DC.W    4       number of priorities
        DC.W    350     number of queues
        DC.W    32      number of alarms
        DC.W    4       number of partitions
        DC.W    0       number of resources
        DC.W    10      tick length
        DC.W    0       ticks for time slice
        DC.L    0       no user clock isr


* Task Initialization Structure for the sample protocol task

        DC.W    SPSTSK_ID       task ID
        DC.W    2               task priority
        DC.L    _spstsk         entry point address
        DC.L    SPSSTKTOP       initial stack pointer
        DC.W    0               time slice enabled
        DC.W    0               filler (not used)


* Task Initialization Structure for the spshio (utility) task

        DC.W    SPSHIO_ID       task ID
        DC.W    2               task priority
        DC.L    _spshio         entry point address
        DC.L    STKTOP_HIO      initial stack pointer
        DC.W    0               time slice enabled
        DC.W    0               filler (not used)

* end of list
        DC.W    0               end of list marker
```

Figure 4–5: Sample Configuration Table with Task Initialization Structures

4.2.3 Task Initialization Routine

A task initialization routine is supplied to be used at the start-up of the ICP. The task initialization routine is executed at the completion of the download sequence and performs the following functions:

1. Load the configuration table address into register A0.
2. Loads the operating system initialization entry point address into register A1.
3. Jumps to the operating system initialization entry point "osinit."

4.2.4 OS/Protogate Initialization

Once the task initialization routine passes control to "osinit", the following operations are performed:

1. Initialize system stack pointer, exception vector table, and clock interrupts (using the tick length specified in the configuration table).
2. Build data structures (task control blocks, queue control blocks, and so on) according to parameters specified in the configuration table.
3. Allocate space for the timer task's stack and create the task.
4. Use the task initialization structures included in the configuration table to create one or more application tasks.
5. Transfer control to the kernel's dispatcher to begin normal run-time operations.

The timer task is the highest priority in the system and is dispatched first. It performs certain initialization procedures and then stops, after which the other tasks that were created are dispatched in order of priority.

4.3 Determining Configuration Parameters

Although the design of a system should never be constrained by its configuration, when available memory is extremely limited or system performance is critical, it might be wise to consider the relationship between the system configuration and OS/Protogate's memory requirements and performance. These relationships are discussed in the following sections.

4.3.1 OS/Protogate Memory Requirements

OS/Protogate requires memory space for code, system data, stacks, and the exception vector table. Some data requirements are fixed, and some are dependent on the system configuration. The space required for the exception vector table, code, and fixed data for a particular version of the operating system can be found in `osp_2432B.map` for `osp_2432B.mem`. The number of bytes required for the system stacks and configurable data structures can be calculated as shown in Table 4–1.

Table 4–1: System Data Requirements

Stack	Bytes Required
Supervisor stack	1024
Timer task's stack	512
Task control blocks	Number of tasks x 24
Queue control blocks	Number of queues x 20
Partition control blocks	Number of partitions x 28
Resource control blocks	Number of resources x 16
Alarm control blocks	Number of alarms x 28
Task alarm control blocks	Number of tasks x 28
Dispatch queues	((Number of priorities + 1) x 8) + 4

Table 4–2, which is based on the configuration shown previously in Figure 4–4 on page 53, shows a sample calculation used to determine the total number of system data bytes required. The total memory requirements for the system are calculated by adding

the total number of system bytes required to the ending address of the system services module and rounding up, if necessary, to an even multiple of four bytes.

Table 4–2: Sample Calculation of System Data Requirements

Stack	Bytes Required		
Supervisor stack			1024
Timer task's stack			512
Task control blocks	8 x 24	=	192
Queue control blocks	30 x 20	=	600
Partition control blocks	4 x 28	=	112
Resource control blocks	0 x 16	=	0
Alarm control blocks	10 x 28	=	280
Task alarm control blocks	8 x 28	=	224
Dispatch queues	((5 + 1) x 8) + 4	=	52

			2996
			or
			0xBB4

4.3.2 Configuration and System Performance

The following fields of the configuration table define the number of control structures to be allocated during system initialization:

`cf_ntask`	Task control blocks and task alarm control blocks
`cf_nque`	Queue control blocks
`cf_nalarm`	Alarm control blocks
`cf_npart`	Partition control blocks
`cf_nresrc`	Resource control blocks

As described in Section 4.3.2.1, the values of these fields, no matter how large, have no effect on system performance. The `cf_nprior` field determines the number of task priorities in the system and affects performance as described in Section 4.3.2.2. The `cf_ltick` field determines the length of a "tick" and the `cf_lslice` field determines the length of a time slice. The relationships of these fields to system performance are discussed in Section 4.3.2.3.

4.3.2.1 Number of Configured Task Control Structures

The `cf_ntask` field of the configuration table defines the number of task control blocks to be allocated in the system. Task control blocks are allocated sequentially, forming an array of structures. The task ID is used as an index into the array to locate a particular task control block. Therefore, the processing time required to access any task control block is fixed and is not dependent on the number of task control blocks in the system. Likewise, and for the same reason, the number of queue control blocks, alarm control blocks, partition control blocks, and resource control blocks has no effect on system performance.

4.3.2.2 Number of Configured Priorities

The `cf_nprior` field of the configuration table determines the number of task priorities to be defined. A dispatch queue is created for each priority. When the head pointer for

a particular dispatch queue is zero, the queue is empty (in other words, no task is scheduled for execution at that priority). When the head pointer is non-zero, it contains the address of a task control block corresponding to a task that is scheduled for execution at that priority. Whenever a task switch occurs, the system dispatcher tests the head pointer of each dispatch queue, in order of priority, until a non-zero value is encountered, then dispatches the task indicated by the task control block address. Because the dispatch queues are searched sequentially, a large number of priorities can adversely affect system performance. There is no benefit to configuring more priorities than required by the system design.

For example, suppose that a particular system consists of the following tasks:

Task ID	Priority
1	0 (timer task)
2	1 (reserved)
3	2
4	2
5	3

The operation of that system is no different than the operation of a system with the same tasks at the following priorities:

Task ID	Priority
1	0 (timer task)
2	50 (reserved)
3	75
4	75
5	200

The priority of task 5 is no lower in the second system than in the first. The difference between the priorities of tasks 1 and 2 is no greater in the second system than in the first. However, the first system executes more efficiently because it requires the configuration of only three priorities (priority 0 is added automatically for the timer task), and the dispatcher must search a maximum of only four dispatch queues at each task switch, rather than the 201 required by the second system.

4.3.2.3 Tick and Time Slice Lengths

Ticks measure the duration of alarms and the system's time slice period. The `cf_ltick` field of the configuration table specifies the length of a tick (1 to 222 milliseconds).

The length of a tick should be set to the smallest of the following values:

- The minimum duration of any alarm in the system
- The maximum acceptable error in an alarm duration
- The desired time slice duration

Because each tick corresponds to a clock interrupt and involves processing by the clock interrupt service routine, setting the tick length to a smaller value than is actually required results in increased overhead and a degradation in system performance.

The `cf_lslice` field of the configuration table specifies the number of ticks for each time slice. The time slice should be long enough to allow each task adequate processing time before being preempted (in other words, to avoid "thrashing"), but not so long that any task is able to prevent other tasks from executing in a timely fashion. (If no tasks in the system are created with time slicing enabled, the length of the time slice is immaterial.)

Chapter

5 Debugging

The debugging facilities available depend on whether Wind River Systems' or some other cross development environment is being used. This chapter describes the debugging facilities provided.

5.1 PEEKER Debugging Tool

PEEKER is a low-level peek and poke routine stored in the ICP2432B's PROM. To use PEEKER, attach a 9600 b/s terminal directly to the ICP's console port with a standard DB-9 to 10 pin box connector cable. To enter PEEKER, type Control-C on the ICP's console device, depress the NMI switch on the ICP2432B's card near the top edge or execute a `trap #15` in your code.

On entry, PEEKER displays the current values of the ColdFire®'s register set.

PEEKER allows you to examine and modify locations in the ICP's memory space by bytes, words, or longwords.

In response to PEEKER's prompt (`pk>`), enter Control-X to return to PEEKER's caller or enter the hex address of a location to examine or modify it.

To examine a location, enter:

- the location's address in hexadecimal
- the access width (preceded by a semicolon):
    - `b` for byte

  - `w` for word
  - `l` for a longword
- an equal sign

PEEKER then displays the address and contents of the given address in the form specified. The data may be modified by entering the new hexadecimal value followed by "^", "=", a space, or a return as listed below.

^	Close current location, open previous location (in address space), and display contents
=	Close current location, open current location (in address space), and display contents
space	Close current location, open next location (in address space), and display contents
return	Close current location and return PEEKER to its initial state, waiting for a new address or Control-X

The following is a typical example:

```
pk> 1000;b=
0000.1000 01 n <return>
0000.1001 10 p <return>
0000.1000 01 <return>
pk>
```

PEEKER uses the following special characters to navigate and/or process inputs:

b	Open by byte
circumflex (^)	Close current location, open previous location (in address space), and display contents
comma	Field delimiter between address and data
Control-X (exit)	Return to whomever called PEEKER
delete	Return PEEKER to its initial state
equal sign	Close current location, open current location (in address space), and display contents

l	Open by longword
linefeed Control-J	Close current location, open next location (in address space), and display contents
space	Close current location, open next location (in address space), and display contents
n (next)	Close current location, open next location (in address space), and display contents
p	Close current location, open previous location (in address space), and display contents
period	Ignore, but echo
r or R	Publish registers and return PEEKER to initial state
<return> <esc>	Close and return to initial state
u (up)	Close current location, open previous location (in address space), and display contents
underscore	Ignore, but echo
w	Open by word (default)

When PEEKER is entered, a brief summary of the special characters is published after the register dump:

```
Peek & Poke <address>[,<data>][;<b, w or l>]<p, =, n, or <return>>
R/r = dump registers
ctrl/x = return to caller
```

The ICP2432B has "reset" and "abort" (NMI) pushbuttons on its circuit board. Pushing the NMI button allows you to break out of loops and gain control even if the CPU is at level seven.

Note

If the vector table entry for Autovector 7 or the vector base register has been corrupted, the result of pushing the NMI button is indeterminate.

5.2 SingleStep Debugging Tool

The *SingleStep Debugger for the ColdFire® Microprocessor Family* manual describes how to use the SingleStep debugging tool provided by Wind River Systems (WRS). SingleStep is a symbolic debugger that allows developers to debug optimized C code for ColdFire® target systems. The debugger can interface with the ICP in two ways: Via the monitor, which is loaded into the ICP SDRAM along with the OS and the application code and the "serial printer" port on the ICP; or via the Background Debug Mode (BDM) connection (26 pin header on the ICP). The BDM method is the more powerful and is recommended.

VMS users must have a PC running DOS and a utility to transport files from the PC to the VMS system.

Modules built with WRS development tools can be downloaded to the ICP along with the WRS RAM-based debug monitor. This monitor runs on the ICP and communicates with SingleStep through one of the ColdFire®'s UARTs. You must connect the UART by a cable from the ICP's "serial printer" port to a serial port on the SingleStep host machine. SingleStep instructs the monitor to set breakpoints, dump memory, view registers, and so on.

You must perform the following basic operations to use SingleStep monitor:

1. In the `spsload` file, uncomment (remove the pound sign) the `LOAD` command for the debug monitor.

2. Install cables that connect the serial port on the SingleStep host machine with the "serial printer" port on the ICP.

3. Reboot the Freeway server or rerun `icpload` on the embedded product to download the SPS software and SingleStep monitor to the ICP.

Once the cables have been properly installed, launch SingleStep on you Windows machine.

Configure the serial communications option for 9600 baud 8 bits, no parity and NO flow control. Be sure to copy the `sstep.ini` and the `MCF5407.cfg` files from the /user/freeway/.../icp2432b directory. Select either `sps.lo` or `sps.los` as the debug file.

Please read the `READ ME file delivered in the above icp2432b directory`

Consult the *SingleStep Debugger for the ColdFire® Microprocessor Family* manual for complete instructions on commands, aliases, and so on.

5.3 System Panic Codes

Protogate's OS/Protogate system software generates an illegal instruction trap (using the `ILLEGAL` instruction) when it encounters a non-recoverable error condition. Before executing the `ILLEGAL` instruction, the operating system stores a "panic code" in the `gs_panic` field of the global system table. The format and location of the global system table is described in the *Freeway OS/ProtogateProgrammer's Guide*, and Appendix A in that document describes the OS/Protogate panic codes.

XIO pushes its panic code onto the stack and calls `hio_panic`, which executes an illegal instruction. The illegal instruction will then trap to PEEKER or the SingleStep monitor. User applications can handle error conditions in the same manner to their own assembly language panic routine.

Chapter

6 ICP Software

6.1 ICP-resident Modules

The ICP-resident sample protocol software (SPS) is downloaded in addition to the system services module. The `sps_fw_2432B.mem` module contains the task-level code and interrupt service routines.

Functionally, the sample protocol software is composed of the protocol and utility tasks and a group of interrupt service routines. Figure 6–1 shows a block diagram of the Freeway server and Figure 6–2 shows a block diagram of the Freeway embedded product.

6.1.1 System Initialization

As the last step of the SPS download (Section 4.1 on page 45), the system is initialized at the address of a system task initialization routine that is part of the SPS module. The task initialization routine loads the address of the system configuration table into register A0 and jumps to OS/Protogate's initialization entry point (osinit). The SPS task initialization routine and configuration table, described in Section 4.2 on page 50, are located in the `spsasm.asm` file located in the `freeway/icpcode/proto_kit/src` directory.

OS/Protogate's osinit routine initializes the operating system variables and data structures, then creates the timer task and the tasks specified in the configuration table. These are the protocol task (`spstsk`) and the utility task (`spshio`). Section 4.2.4 describes the `osinit` procedure in more detail.

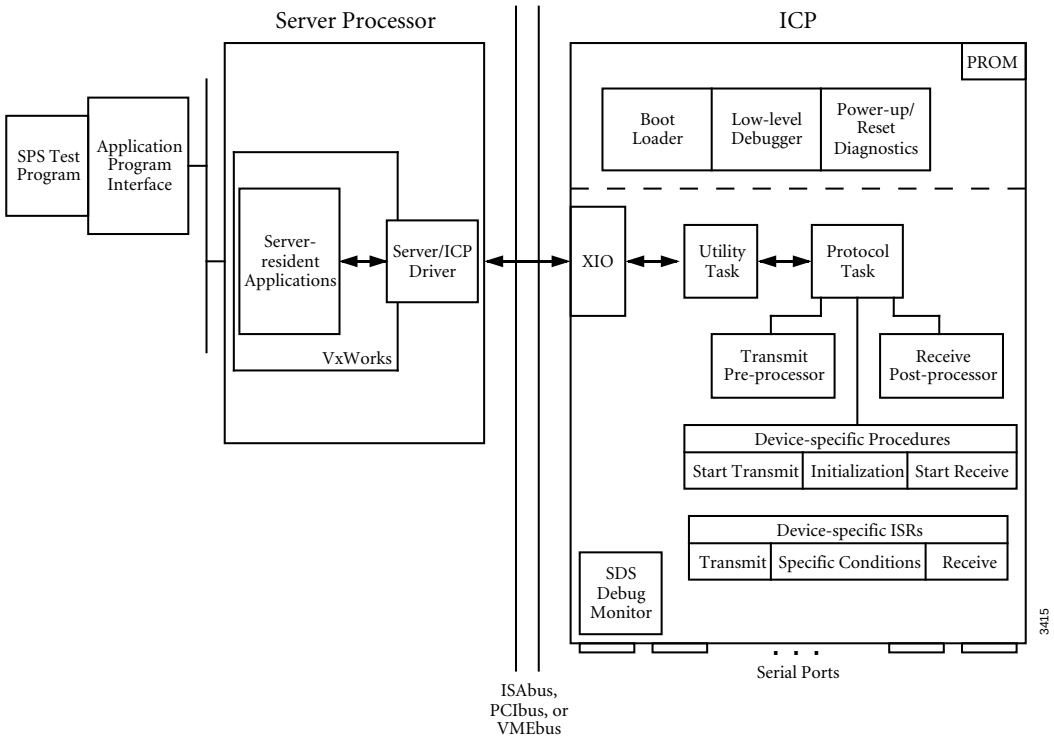


Figure 6–1: Block Diagram of the Sample Protocol Software - Freeway Server

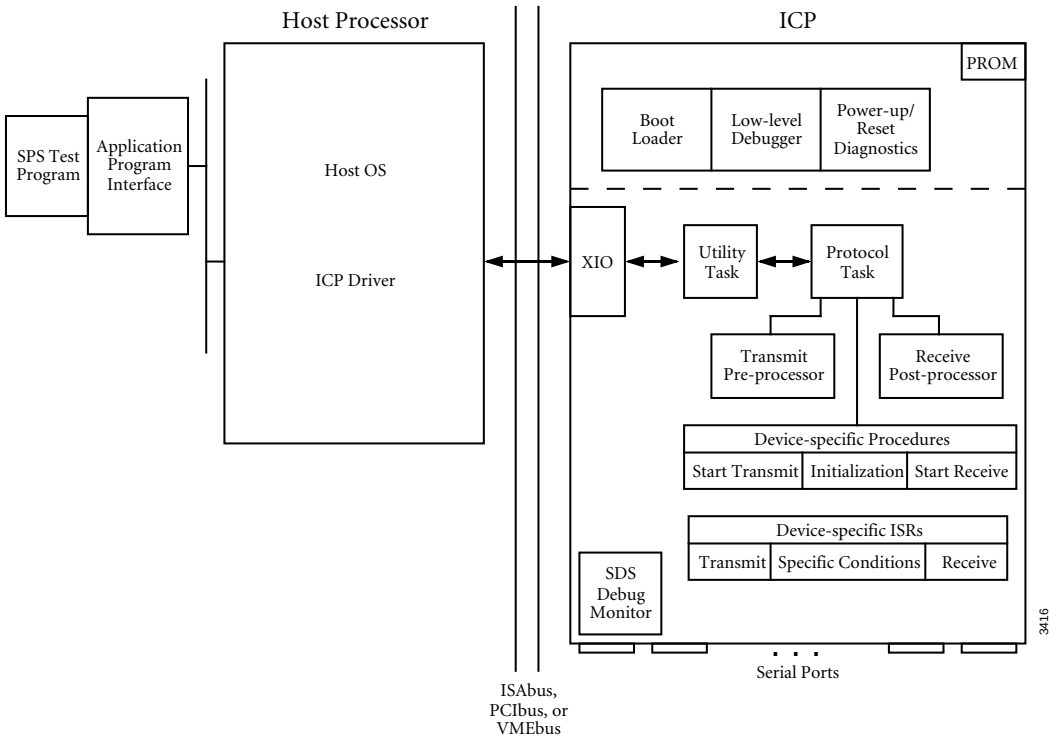


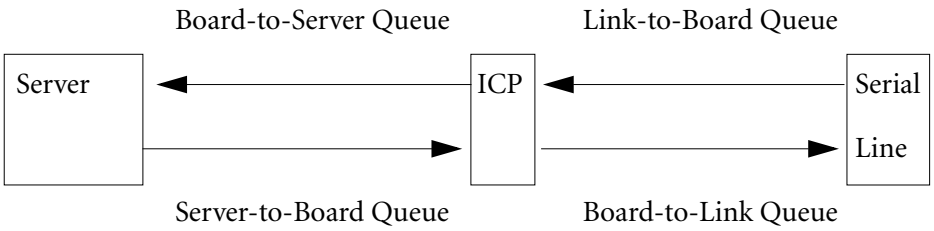
Figure 6–2: Block Diagram of the Sample Protocol Software - Freeway Embedded

6.1.2 Protocol Task

This section explains the buffer management method for writing to or reading from the ICP's host. The eXecutive Input/Output (XIO) interface is a collection of function calls that are executed in the context of the user's application tasks. XIO uses queues that are declared by the utility task.

XIO consists of simple function calls. Section 7.4 on page 108 gives details of XIO.

During its initialization, the protocol task creates queues for each link, which relate to the stages and direction of data flow as follows:



After initialization completes, the protocol task operates in a loop. Within the loop, it makes a series of subroutine calls for each link. In the `chkhio` subroutine, the protocol task checks for messages from the ICP's host that have been routed to the individual queues by the utility task; these messages are then processed according to command type. For a transmit data block command, the message is not processed immediately, but is transferred to the link's board-to-link queue, where it is later processed in the `chkloq` subroutine.

In the `chkloq` subroutine, which is called only for active links, the protocol task sends data buffers associated with completed transmit messages back to the application program as write acknowledgments and checks the board-to-link queue for transmissions that are ready to be started.

In the `chkliq` subroutine, also called only for active links, the protocol task checks the link-to-board queue for buffers that have been filled with received data at the interrupt

level. Completed received data messages are sent to the link's board-to-server queue to await processing by the utility task.

When all links have been processed, the protocol task suspends. It continues when a message is posted to any of its queues or when an interrupt service routine notifies it that a transmit or receive operation has completed. The interface between the protocol task and its interrupt service routines is described in Section 6.2.

The SPS utility task, `spshio`, sets up an interface to XIO during its initialization, then enters a loop. Within the loop, it checks its input queue for returned header buffers as well as messages from the ICP's host that have arrived on node 1 and node 2. It also checks the protocol task's board-to-server queues for messages to be sent to the host. It then suspends, and will be unsuspended by the protocol task or when a message is posted to its input queue. The operation of the utility task is described more completely in Section 6.1.3.

6.1.3 Utility Task (spshio)

The ICP-resident software communicates indirectly with the ICP's host through the part of the system services module called the XIO interface. The utility task, `spshio`, handles the interface between the protocol task, `spstsk`, and XIO. This section describes the utility task and its relationship with the protocol task. Chapter 7 provides a more detailed explanation of the ICP/host protocol used for communication between the utility task and XIO.

As described in Section 6.1.2, the protocol task, `spstsk`, creates an board-to-server queue and a server-to-board queue for each link during its initialization. These queues hold messages to be transferred to and from the ICP's host by the utility task. (Section 7.2.3 on page 97 describes the node declaration queues.) The protocol task is also responsible for creating the buffer partition that contains data buffers to be used for passing data to and from the ICP's host. The size of the buffers created for this par-

tition depends on the value of the `buffer.size` file which is downloaded with the application. (See the `/freeway/boot/spsload` file.)

During initialization, the utility task creates the header buffer partition and posts node declaration queue requests to XIO to establish nodes to be used by the ICP for reading from, and writing to, the ICP's host. As requested by the utility task, XIO creates read and/or write request queues for each node. Node 1 (the main node) and node 2 (the priority node) are special insofar as all information coming to the ICP from the ICP's host arrives through these nodes. These nodes do have write queues, and in rare cases (such as rejecting an erroneous attach request) are used to pass information back to the ICP's host, but for the most part they are a one-way path for messages coming from the ICP's host. These messages are then de-multiplexed to the various links. The remaining nodes are used strictly by the ICP for writing to the ICP's host.

The utility task begins by creating all the nodes as well as the queues for the system header and data buffers. After this initialization, the utility task operates in a loop and performs the following functions:

1. Keeps reads posted on the main and priority nodes
2. Distributes incoming buffers to the correct server-to-board queues
3. Posts buffers from the board-to-server queues to the appropriate nodes

The utility task is also responsible for the verification of session and link IDs, and for swapping bytes within words (to allow for differences in word ordering for Big Endian (Motorola) and Little Endian (Intel and VAX)), both for messages coming from and messages going to the ICP's host. When no message processing is required, the utility task suspends and will be unsuspended by the protocol task or when a message is posted to its input queue.

The following sections provide detailed examples of read and write processing by the utility task. Figure 6–3 shows the SPS message format.

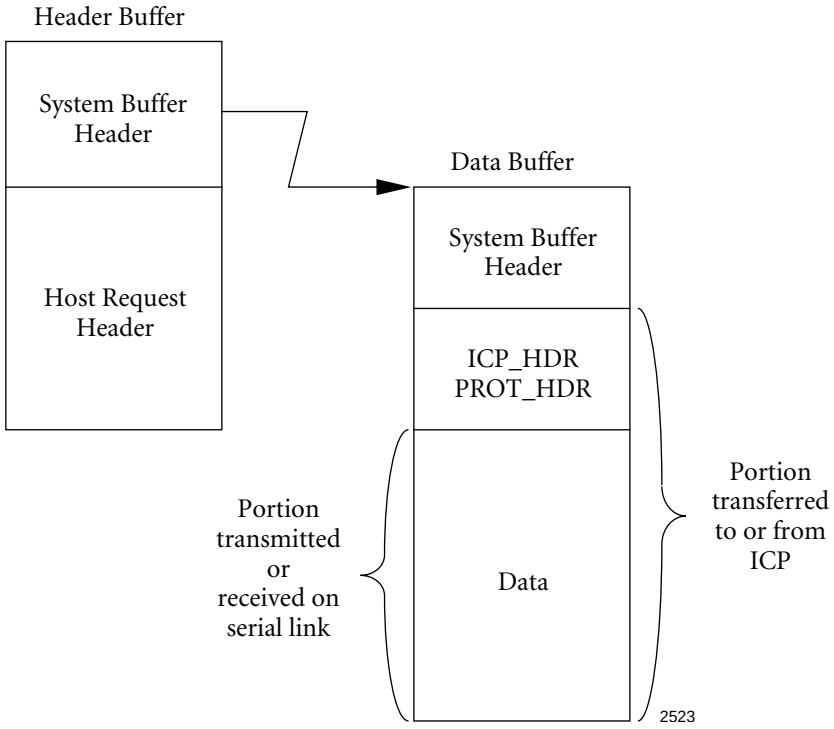


Figure 6–3: Sample Protocol Software Message Format

6.1.3.1 Read Request Processing

The utility task, `spshio`, issues read requests to XIO to obtain messages from the ICP's host, which could be either data or control messages. A message from the ICP's host contains one of the command codes described in Section 8.5. The `DLI_PROT_SEND_NORM_DATA` command code is used as an example in this section to describe the steps involved in processing read requests. Figure 6–4 illustrates these steps.

1. To obtain messages from the ICP's host, the utility task creates read request queue elements composed of headers from partition H and data buffers from partition D. The utility task sets the disposition flags in the system buffer headers to inform XIO of the action it should take when the request is complete. It also sets the node number in the host request header for XIO to use in communicating with the host. Sixteen queue elements are created for node 1 and sixteen for node 2. These are the only nodes to which the host can write.

2. The utility task issues read requests to XIO for each queue element created in Step 1.

3. For each read request, XIO posts a read to the Read Request Queue associated with the node identified in the host request header.

4. When the ICP's host sends a write request to its driver, XIO transfers the message to the data buffer, and the ICP read request issued in step 2 is complete.

5. XIO posts the header and the data buffer to the utility task's data and header input queues for node 1 or 2.

6. The protocol and utility tasks then do the following:

    a. Based on the `session` or `link` field of the ICP header, the utility task multiplexes and transfers the data buffers from its data input queue to the appropriate server-to-board queue.

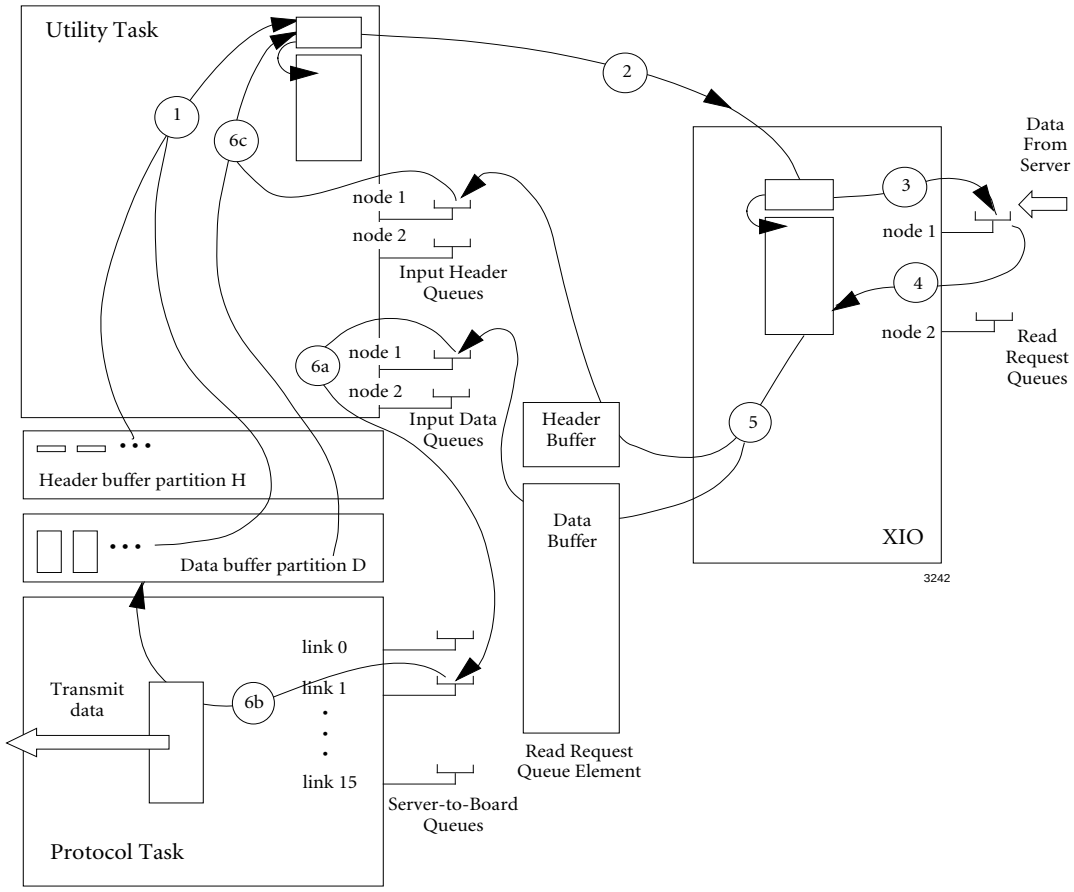


Figure 6–4: ICP Read Request (Transmit Data) Processing

b. The protocol task removes data buffers from the server-to-board queue, processes the requests, then releases the buffers to partition D or uses them to send acknowledgments back to the application program.

c. The utility task obtains additional data buffers from partition D and links them to header buffers that were returned to its header input queue. It then issues new read request to XIO for node 1 or 2 (depending on the node from which the header buffers were returned). In this way, the utility task attempts to keep at least one read request pending at all times.

6.1.3.2 Write Request Processing

The utility task issues write requests to XIO when data is received on a serial line or in response to other requests from the ICP's host. A message to the ICP's host can contain a received data block, a statistics report, an error message, or some other acknowledgment to a client application program. A received data block is used as an example in this section to describe the steps involved in processing write requests. Figure 6–5 illustrates these steps.

1. The protocol task obtains a data buffer from partition D, to be filled with data received on a particular link. When a block of data has been received, the protocol task posts the buffer to the link's board-to-server queue.

2. When the utility task finds the data buffer on the board-to-server queue, it links the buffer to a header buffer obtained from partition H, creating a write request queue element. The utility task sets the disposition flags in the system buffer headers to inform XIO of the action it should take when the request is complete. It also sets the link's previously assigned node number in the host request header for XIO to use in communicating with the host.

3. After filling out the `data length` and `session` fields of the ICP and PROT headers, the utility task issues the write request to XIO.

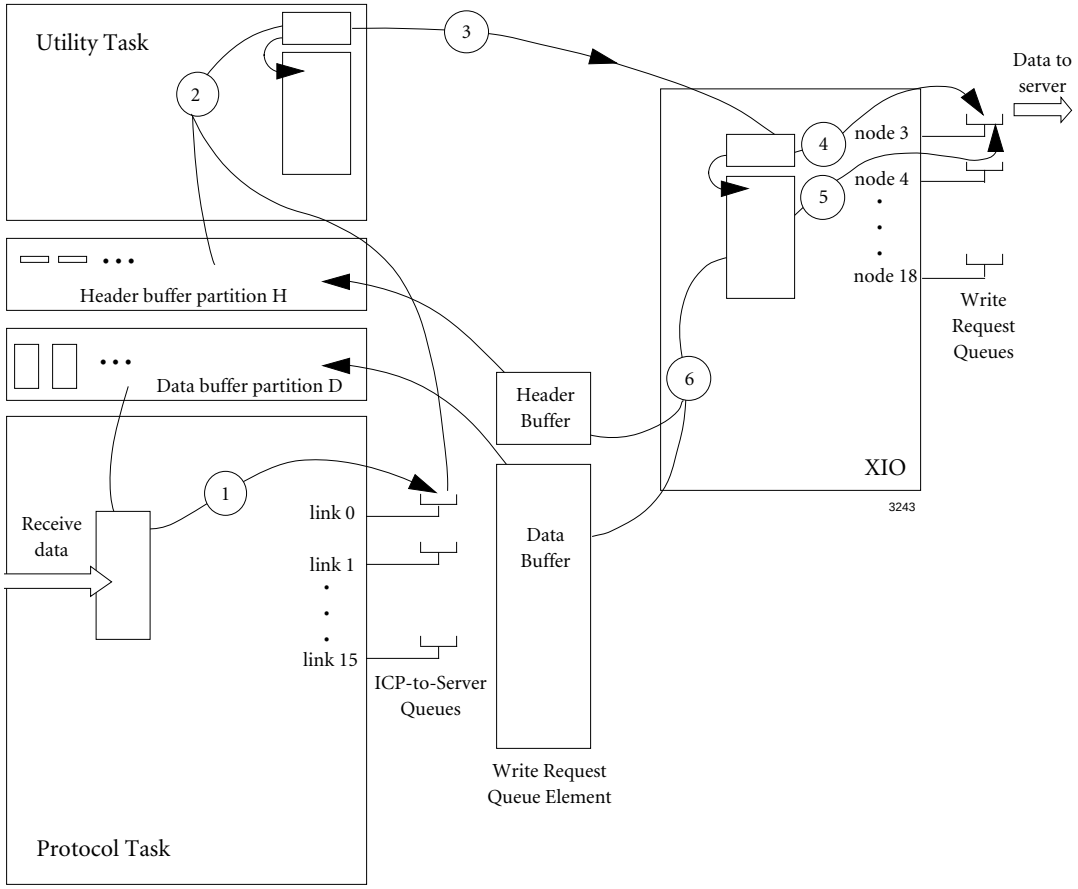


Figure 6–5: ICP Write Request (Receive Data) Processing

4. XIO posts a write to the Write Request Queue associated with the node identified in the host request header.

5. When the ICP's host sends a read request to its driver with a matching node number, XIO transfers the message from the data buffer to the ICP's host memory and the ICP write request issued in step 3 is complete.

6. As instructed by the disposition flags, XIO releases the header and data buffers to their respective partitions.

6.2 Control of Transmit and Receive Operations

Various techniques are available for coordinating transmit and receive operations at the task and interrupt level. The simplest method is to start every operation from the task level. In this case, a signal of some kind must be sent from the interrupt service routine to the task level at completion, at which time the task can start the next operation. This is the method used by the SPS for data transmissions.

Another option is to maintain a queue of messages. To save time in the interrupt service routine, messages can be added to the tail and removed from the head of the queue at the task level, with the interrupt service routine moving from message to message within the queue using a `link` field in the buffer headers. An example of this technique is provided by the SPS receive operations.

The following sections describe the task/interrupt-service-routine interface used to control transmit and receive operations for the SPS.

6.2.1 Link Control Tables

The protocol and utility tasks and the interrupt service routines communicate and coordinate their operations for each link by means of a global link control table. One link control table is allocated for each link. The link control table contains state information, queue IDs, configuration parameters, IUSC register values and/or addresses, transmit and receive control parameters, configuration-specific subroutine addresses, statistics information, and so on. The link control table is defined in `/freeway/icpcode/proto_kit/src/spsstructs.h`. Please review it with your favorite editor before reading the following discussions.

6.2.2 SPS/ISR Interface for Transmit Messages

When the protocol task receives a transmit data block message on a link's server-to-board queue, it moves the message to the link's board-to-link queue to await transmission. The board-to-link queue is processed in the `chkloq` subroutine according to the mode of communication.

The `lct_flags` field in the link control table is cleared by the protocol task when it initiates a transmission and is set by the interrupt service routine when the transmission is finished. A transmission can be initiated only when the link is in the IDLE state. The protocol task points the transmit data block message on the head of the board-to-link queue, calls the appropriate preprocess routine for the protocol to prepare the data for transmission, and calls the subroutine `xmton` to set up the hardware devices for transmission of the data. `Xmton` clears the `flags` field in the buffer's headers and clears the `lct_flags` and states in the link control table. When the transmit completes, the interrupt service routine sets `flags` in the buffer's headers, initializes the `lct_flags` and states in the link control table, and resumes the protocol task. The protocol task releases the completed buffer and starts the transmission of the next message on the queue.

6.2.3 SPS/ISR Interface for Received Messages

When a link is enabled, the `rcvstr` subroutine for the requested protocol (located in `asydev_iusc.c`, `bscdev_iusc.c`, and `sdlcdev_iusc.c`) is called, which calls "restock" to preallocate data buffers from the data buffer partition for posting to the link-to-board queue. The `lct_frbuf` field in the link control table is set to the address of the first buffer on the queue.

When a frame is received, the buffer is filled, and the interrupt service routine updates `lct_frbuf` to the next buffer on the queue using the `sb_nxte` field in the system buffer header. (The interrupt service routine does not unlink the filled buffer from the queue).

In the `chkliq` subroutine, the protocol task determines whether the buffer at the head of the queue has been completed (a block has been received). If the receive is finished,

and the protocol task removes the buffer from the link-to-board queue, calls the appropriate postprocessor to process the data before passing it to the application program, posts it to the board-to-server queue, and resumes the utility task which passes the message to the host.

Whenever the protocol task removes a buffer from the head of the link-to-board queue, it restocks the queue. In this way, the protocol task maintains several available buffers for received messages.

Figure 6–6 shows a link-to-board queue containing four buffers. Two are filled and waiting for removal by the protocol task. The third buffer is set up for the current receive.

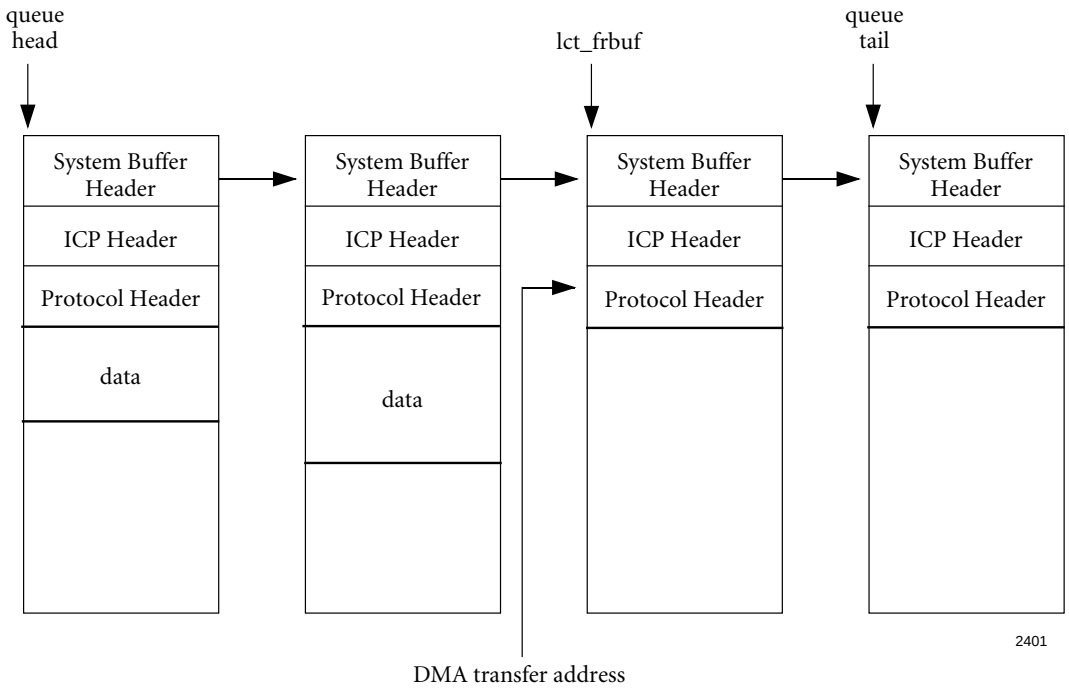


Figure 6–6: Sample Link-to-Board Queue

6.3 Interrupt Service

At the interrupt level, the SPS provides specific examples of IUSC programming for asynchronous (ASYNC), byte synchronous (BSC), and bit synchronous (HDLC/SDLC) modes of operation. At the same time, examples are provided for:

- operation with and without the use of DMA
- C and assembly language programming
- CRC calculation in hardware (by the IUSC) or in software

Table 6–1 summarizes these features for each mode of operation.

Table 6–1: Summary of Communication Modes

	Asynchronous	**BSC**	**HDLC/SDLC**
IUSC mode	Asynchronous	Byte synchronous	Bit synchronous
Data transfer method	Character interrupts	Character interrupts/DMA	DMA
Start block detection (receive)	ISR search for start character	IUSC detects SYNC character	IUSC detects opening flag
End block detection (receive)	ISR search for end character	Byte count in header	IUSC detects closing flag
CRC calculation	Software	Software	IUSC
ISR programming language	C	Assembly/C	C

6.3.1 ISR Operation in HDLC/SDLC Mode

In HDLC/SDLC mode, DMA is used for both transmit and receive. The IUSC automatically provides the opening and closing flags on transmit. The DMA transfer count is set to the number of bytes in the frame, not including CRC and flags. The IUSC is set to calculate the CRC during transmission of the frame and to send the CRC when it detects a transmit underrun. When the DMA reaches terminal count (and no longer

transfers characters to the IUSC), a transmit underrun is generated. The IUSC transmits the two-byte CRC followed by a closing flag to terminate the frame.

To receive, the DMA transfer count is set to the maximum block size and will not normally reach terminal count. The IUSC automatically calculates CRC during the received frame and generates an end-of-frame (special receive condition) interrupt when the closing flag is detected. The interrupt service routine reads an SCC register to determine whether the CRC that the SCC calculated matched the CRC bytes received at the end of the frame. The IUSC posts the status of the reception in the linked list header record (located at the end of the data buffer) which is then examined by the ISR.

The following interrupts are processed in HDLC/SDLC mode:

IUSC End of Buffer If terminal count is reached before end-of-frame, the received message is too long (receiving more data would overrun the receive buffer). In this case, the interrupt service routine increments an error count and restarts the receiver using the current receive buffer.

IUSC RDMA Complete This interrupt is generated at the end of a received frame. If the IUSC indicates a CRC error, an error count is incremented, and the receiver is restarted using the current buffer. If the CRC is good, the receiver is restarted using the next buffer in the link-to-board queue.

IUSC End of Buffer This interrupt is enabled only by the external or transmit status interrupt service routine when a transmit underrun occurs while the transmit buffer is not yet empty. The end of the transmission is processed.

Loss of DCD An error count is incremented and the receiver is restarted using the current receive buffer.

Abort An error count is incremented and the receiver is restarted using the current receive buffer.

Transmit Underrun If the DMA has reached terminal count, transmit underrun can cause an external/status interrupt. This indicates end-of-frame on transmit, although the final character of the frame might not yet be completely sent. If the SCC transmit buffer is empty, the end of the transmission is processed. Otherwise, the SCC's transmit interrupt is enabled, so the end of the transmission can be processed when the transmit buffer becomes empty. If a transmit underrun interrupt is generated when the DMA has not reached terminal count, an actual underrun has occurred. An error count is incremented, but the transmission is allowed to continue. (The receiving link detects a CRC error on the frame.)

6.3.2 ISR Operation in Asynchronous Mode

For asynchronous mode, DMA (conditional compile option for the IUSC) is not used for either transmit or receive. Rather, the IUSC is set up to generate interrupts on every character received and transmitted. On transmit, a count is decremented as each character is written to the IUSC's transmit buffer, and the block is complete when the count reaches zero. On receive, user-configured start and end characters are used to delimit a block. CRC, if enabled, is calculated and compared at the task level.

The following interrupts are serviced in asynchronous mode:

IUSC Receive Character Available This interrupt is generated on every received character. The receive interrupt service routine is state-driven. After transferring the received character from the IUSC receiver to the receive data buffer, the interrupt service routine processes the character according to the current state:

State 0 Search for start character. If the start character is found, move to state 1; otherwise, take no action and ignore the current character (it will be overwritten by the next character).

State 1 Receive frame. Check for stop character. If the stop character is found and CRC is enabled, move to state 2. If the stop character is found and CRC is

not enabled, process the end of received block and restart the receiver at state 0 using the next buffer in the link-to-board queue. If the stop character is not found, store the character and increment the count.

State 2 First CRC byte. Move to state 3.

State 3 Second CRC byte. Process the end of received block and restart the receiver at state 0 using the next buffer in the link-to-board queue.

IUSC Receive Status This interrupt is generated on receiver overrun, parity error, or framing error. The appropriate error count is incremented, but the receive is not aborted.

IUSC Transmit Buffer Empty This interrupt is generated on every transmitted character. The transmit byte count is decremented, and the end of the transmission is processed if the count has reached zero. (The next transmission is started at the task level.) The IUSC is set up to interrupt when there are 16 bytes free in its transmit FIFO and up to 16 bytes are loaded during each interrupt.

6.3.3 ISR Operation in BSC Mode

In BSC mode, a simple header is prepended to the start of the data block, containing a user-configured start character and a byte count. For transmit, the DMA transfer count is set to the number of bytes in the block, including the header and the two-byte CRC, if enabled. The CRC is calculated and appended to the data at the task level.

For receive, the IUSC is initially set up to generate interrupts on every character received. Each character is compared to the configured start character. Once the start character has been found, the remainder of the BSC header can be received. The DMA transfer count is set to the value specified in the BSC header, IUSC receive interrupts are disabled, and DMA is used to receive the remainder of the message.

The following interrupts are processed in BSC mode:

IUSC Receive Character Available While enabled, this interrupt is generated on every received character. No data is transferred to the receive buffer until the data count is received. When the entire three-byte header has been received, the interrupt service routine disables receive and special receive condition interrupts, sets the DMA transfer count according to the count field of the BSC header (plus two if CRC is enabled), and initiates DMA transfer.

IUSC Special Receive Condition While enabled (before and during reception of the BSC header), this interrupt is generated on receiver overrun errors. An error count is incremented and the receive is aborted.

IUSC End of Buffer This interrupt is generated when the data portion of a BSC message has been received. The interrupt service routine re-enables IUSC receive and special receive condition interrupts and restarts the receiver using the next buffer in the link-to-board queue. CRC, if enabled, is checked at the task level.

IUSC End of Buffer This interrupt is generated at the end of a transmitted frame. The interrupt service routine processes the end of the transmission. (The next transmission is started at the task level.)

Chapter

7 Host/ICP Interface

This chapter describes the interface between the ICP's host processor and an ICP. This interface will be referred to as the host/ICP interface. It is managed by an XIO interface which runs on the ICP, in the OS/Protogate environment, and provides a queue-driven, non-blocking interface to the host processor. Section 7.4 on page 108 gives details of XIO.

7.1 ICP's Host Interface Protocol

Communications between the ICP's host and the ICPs is performed by the host's driver, `icp.c`, and the ICP's driver, XIO. Information concerning any data transfers between the two is passed through a Protocol eXchange Region (PXR).

The PXR for the ICP2432 is implemented via mailboxes within the PCI interface chip. They are accessed as 32-bit entities so that the issue of little or big endian are alleviated.

The ICP2432B is responsible for actually moving the data buffers between the host and the ICP.

When the host has a buffer into which data may be transferred, it issues a "read" request to the ICP along with the address of the buffer and the maximum amount of data it can hold. When the ICP receives a matching request from its application program, it moves the data into the host's buffer and then signals the host that the "read" is complete.

When the host has a buffer of data ready to transfer to the ICP, it issues a "write" request to the ICP with the address of the data's buffer and the amount of data in it. When the

ICP receives a matching request from its application program, it transfers the data and signals the host with a "write" complete.

Since this protocol is asynchronous, the host can send any number of requests to the ICP without waiting for completions on previous requests, and the ICP can process requests and return completions in any order.

The ICP driver and the host driver coordinate the information flowing between the ICP and the host processor by means of node numbers. Each node can have one read and one write queue. At startup, the utility task creates the nodes it will be using, up to the maximum number of nodes allowed by the configuration parameters of the two drivers.

The ICP posts read requests to node 1 (the main node) and node 2 (the priority node); all information coming to the ICP from the host processor arrives through these two nodes. These nodes do have write queues, and in rare cases (such as rejecting an erroneous attach request) are used to pass information to the ICP's host, but for the most part they are a one-way path for messages coming from the host; these messages are then de-multiplexed to the various links. The remaining nodes are used strictly by the ICP for writing to the host. The ICP read request processing is explained in more detail in Section 6.1.3.1 on page 76 and is shown in Figure 6–4 on page 77.

After a message bound for the application program is processed by the protocol task, it is posted to the board-to-server queue belonging to that link. The utility task subsequently removes the message from that queue, prefixes a properly initialized buffer header (for example, providing information on what to do with process completions), then posts it as a write request to a write queue belonging to one of the nodes created at startup. (The particular node is ascertained by indexing into the session table and accessing the node number field by means of the unique session ID that was assigned to that link as a result of a prior attach command.) XIO then passes the message through to the host processor. The ICP write request processing is explained in more detail in Section 6.1.3.2 on page 78 and is shown in Figure 6–5 on page 79.

Note that when a request completion is received from the host processor, XIO uses the node number and request type to match the completion with a pending request. Therefore XIO does not send the host processor a request for a particular node number until any pending request from the same node number is complete. Additionally, requests for any queue are sent to the host processor in the same order they were posted to the queue, but no order is guaranteed for requests posted to different queues. Likewise, notifications of completion are guaranteed to be in the same order that the completions were received from the host processor, but the host processor is not required to send completions in the same order that it received the requests. XIO provides two options for processing the request completions. These options are described in Section 7.2.3.1 on page 99.

7.2 Queue Elements

In general, a queue element consists of one or more linked buffers, and a queue can contain one or more linked queue elements. Every buffer of a queue element contains a standard system buffer header, as defined in the *Freeway OS/ProtogateProgrammer's Guide*. A field in each buffer's header is used as a link to the next buffer of the queue element. Two fields in the header are valid only in the first buffer of a queue element. One field is a link to the next element on a queue and the other, if the queue is doubly linked, is a link to the previous element.

A buffer can be obtained from a system partition (using the get buffer, `s_breq`, system call), but this is not a requirement. Any block of memory large enough to contain a system buffer header can be used as a buffer (for example, a fixed data structure defined within an ICP-resident application). There is no maximum buffer size, no maximum number of buffers in a queue element, and no maximum number of queue elements attached to a queue.

Figure 7–1 shows a singly-linked sample queue containing three queue elements. Figure 7–2 shows a doubly-linked sample queue containing three queue elements.

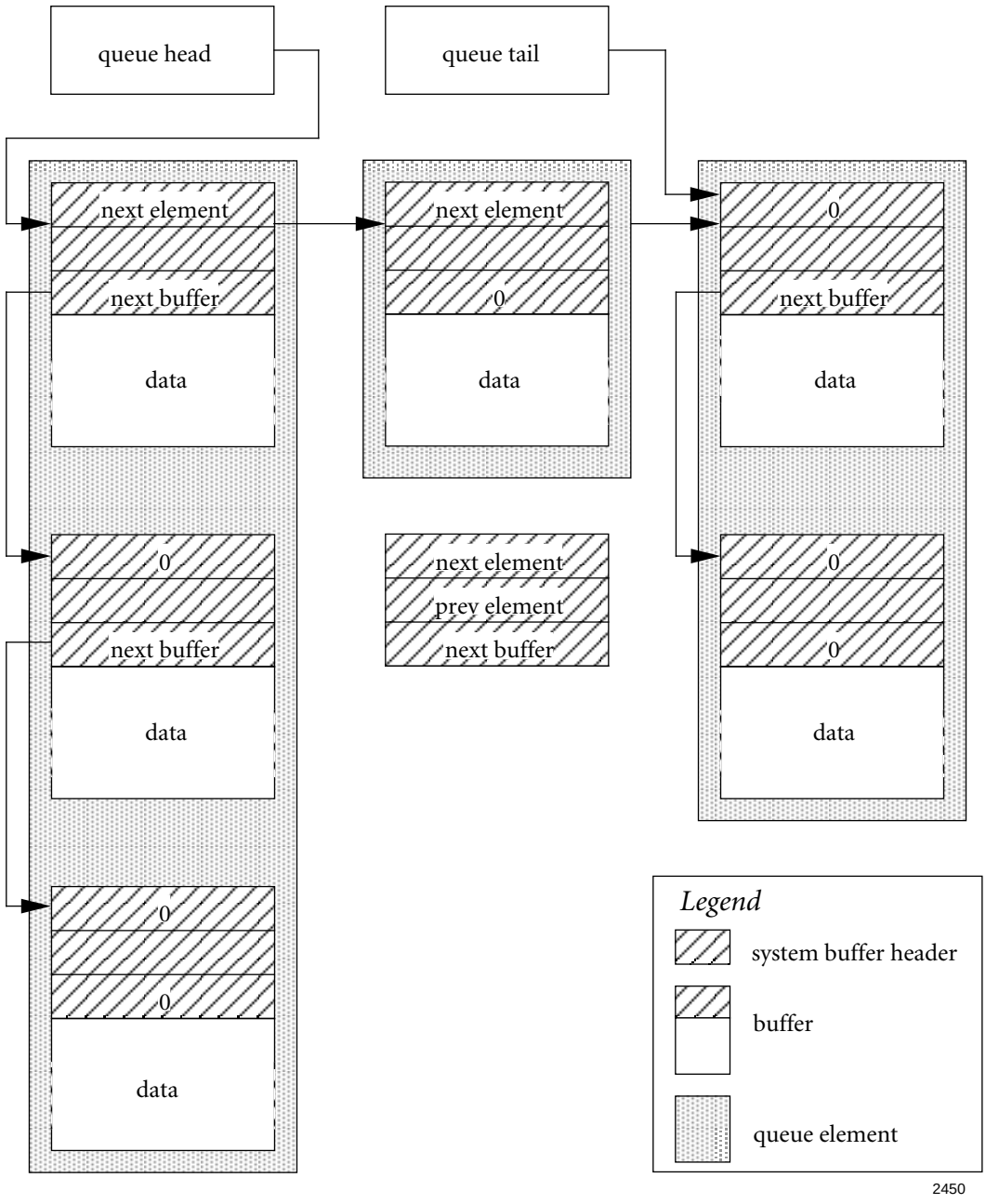


Figure 7–1: Sample Singly-linked Queue with Three Elements

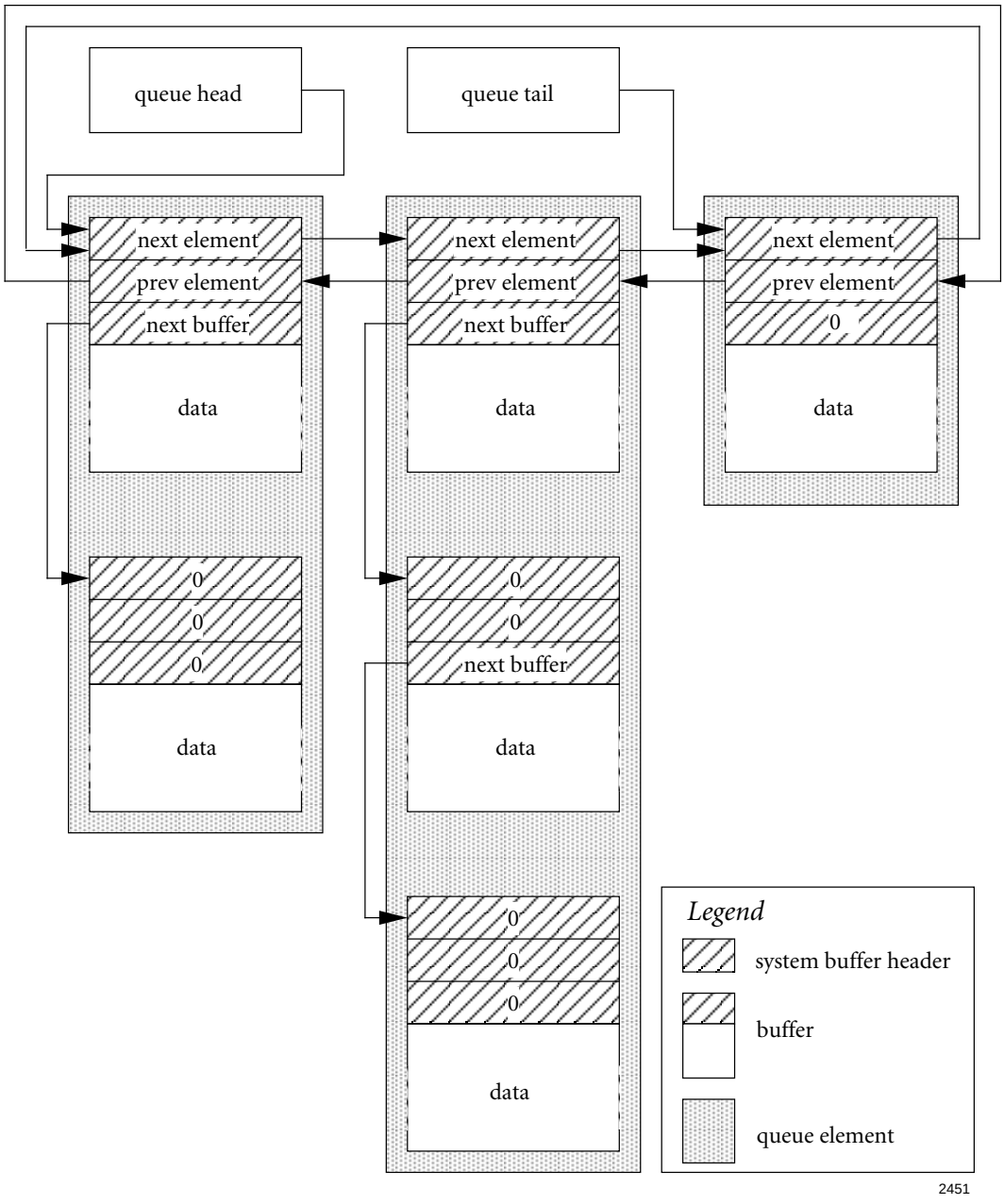


Figure 7–2: Sample Doubly-linked Queue with Three Elements

7.2.1 System Buffer Header

As mentioned previously, every buffer of every queue element must begin with a system buffer header. The following structure defines the format of the system buffer header:

```
struct SBH_TYPE
{
   struct SBH_TYPE *sb_nxte;  /* next element         */
   struct SBH_TYPE *sb_pree;  /* previous element     */
   struct SBH_TYPE *sb_thse;  /* this element         */
   struct SBH_TYPE *sb_nxtb;  /* next buffer          */
   unsigned short   sb_pid;   /* partition ID         */
   unsigned short   sb_dlen;  /* data length          */
   unsigned short   sb_disp;  /* disposition flag     */
   unsigned short   sb_dmod;  /* disposition modifier */
};
```

The header fields, as used by the system, are described below:

Next Element This field is used only by the operating system, and only in the first buffer of a queue element. While the element is attached to a singly- or doubly-linked queue, this field contains the address of the next element on the queue.

Previous Element This field is used only by the operating system, and only in the first buffer of a queue element. While the element is attached to a doubly-linked queue, this field contains the address of the previous element on the queue.

This Element This field is used only by the operating system, and only in the first buffer of the queue element, as a consistency check when the element is posted to or removed from a queue. This field contains the address of the buffer itself (that is, the address of the queue element).

Next Buffer This field contains the address of the next buffer of the queue element. In general, this field must be zero in the last buffer. In the data buffer of a host request queue element, XIO uses this field for a special purpose, as described in Section 7.2.4.1 on page 104.

Partition ID This field contains the partition ID if the buffer was obtained from a partition.

Data Length This field contains the number of valid bytes of data in the buffer (excluding the system buffer header).

Disposition Flag This field, in combination with the disposition modifier, indicates the action to be taken by XIO when processing of the queue element is complete (when the request completion is received from the host). This flag has the following possible values:

`POST_QE`	Post queue element to queue
`FREE_QE`	Zero disposition modifier to mark queue element free
`TOKEN_QE`	Release queue element to a resource
`POST_BUF`	Post buffer to queue
`FREE_BUF`	Zero disposition modifier to mark buffer free
`TOKEN_BUF`	Release buffer to a resource
`REL_BUF`	Release buffer to partition

`POST_QE`, `FREE_QE`, and `TOKEN_QE` are valid only in the first buffer of a queue element and apply to the entire queue element (the

disposition flag is then ignored in all other buffers of the queue element). POST_BUF, FREE_BUF, TOKEN_BUF, and REL_BUF are valid in all buffers and apply only to an individual buffer. For example, in a queue element consisting of only one buffer, POST_QE is equivalent to POST_BUF, but for a multiple-buffer queue element, the value POST_QE in the first buffer indicates that the queue element is to be posted to a particular queue intact, but the value POST_BUF in every buffer indicates that each buffer is to be posted to a queue as an individual queue element. Section 7.2.3.1 on page 99 and Section 7.2.4.1 on page 104 describe the use of this field in more detail.

Disposition modifier This field provides additional information required for completion processing by XIO. What is contained in this field depends on the value of the disposition flag, as follows:

Disposition Flag	Disposition Modifier
POST_QE	Queue ID
FREE_QE	Non-zero value to be cleared
TOKEN_QE	Resource ID
POST_BUF	Queue ID
FREE_BUF	Non-zero value to be cleared
TOKEN_BUF	Resource ID
REL_BUF	Not used

7.2.2 Queue Element Initialization

For the utility task to communicate with the host, it must post at least three node declaration queue elements, described in Section 7.2.3, to XIO's public node declaration queue during its initialization. Two of these, the main node and the priority node, are the conduits for passing information from the host to the ICP. The remaining nodes

are used by your ICP-resident software to send information in the form of data and command acknowledgments to the host processor.

Each node declaration queue element must contain a unique ICP node number and unique queue IDs which will be used for the read and write queues for that node. Once this operation is complete, the utility task can begin posting host request queue elements, described in Section 7.2.4, to these queues. The following sections describe the two types of queue elements.

7.2.3 Node Declaration Queue Element

The utility task, `spshio`, creates node declaration queue elements, generally during initialization, and posts them to XIO. These queue elements identify the ICP node number that the task will use and queue IDs to which either read or write requests (or both) for that node will be posted. (XIO creates the queues. Only the queue IDs are supplied by the utility task.) In your code, you can declare several nodes (up to the maximum allowed by the driver), but you must post a separate node declaration queue element for each.

Since ICP node numbers must be unique throughout an ICP subsystem, a task can declare a node number only once; no other tasks can make a declaration using that node number. The queue IDs associated with declared node numbers must also be unique. A single ID cannot be used as both the read and write queue for a node, nor can it be used for other nodes or for any other purpose.

The node declaration queue element consists of a single buffer containing a system buffer header followed by a node declaration header. The queue element is shown in Figure 7–3 and has the following format:

```
struct NODEC_TYPE
{
   struct SBH_TYPE sbh;    /* system buffer header          */
   unsigned short rqid;    /* host read request queue ID    */
   unsigned short wqid;    /* host write request queue ID   */
   unsigned char  node;    /* ICP node number               */
   unsigned char  status;  /* completion status             */
};
```

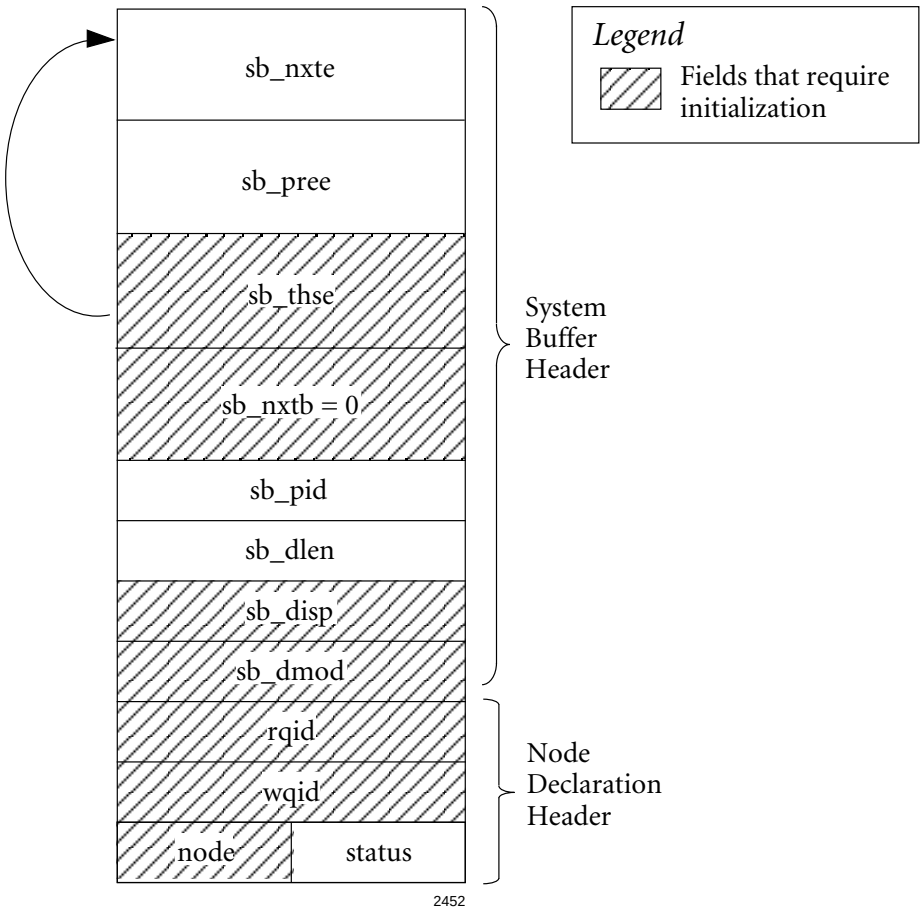


Figure 7–3: Node Declaration Queue Element

7.2.3.1 System Buffer Header Initialization

In the system buffer header, the sb_thse field must be set to the starting address of the buffer. (This field is set by the system if the buffer was obtained from a partition.) The sb_nxtb field must be set to zero. The disposition flag, sb_disp, and disposition modifier, sb_dmod, fields must be initialized as described in the following paragraphs, but no other fields in the system buffer header require specific initialization.

The disposition flag must be set to one of the values defined for the field as described in Section 7.2.1 on page 94. If the requesting task obtained the buffer from a partition, and if it does not require notification when the request has been processed by XIO, the value REL_BUF can be used; this causes XIO to release the buffer to its partition on completion. However, since a post to either of the host request queue IDs specified in the queue element fails if XIO has not yet processed the request (and created the queues), tasks generally request completion notification. Since the queue element consists of only one buffer, POST_QE and POST_BUF are equivalent, and cause XIO to post the queue element to a specified queue. Likewise, FREE_QE and FREE_BUF are equivalent, and cause XIO to clear the disposition modifier. TOKEN_QE and TOKEN_BUF are also equivalent and cause XIO to release the queue element to a specified resource.

If the disposition flag is set to POST_QE or POST_BUF, the disposition modifier must contain a valid queue ID. If the requesting task is the owner of that queue, it then suspends its operation, and resumes when XIO posts the queue element (with a post and resume system call) to the queue on completion.

If FREE_QE or FREE_BUF is specified in the disposition flag, the disposition modifier should be set to a non-zero value, so the requesting task can recognize the completion when the field is cleared by XIO.

If TOKEN_QE or TOKEN_BUF is specified in the disposition flag, the disposition modifier must contain a valid resource ID. The requesting task cannot use this method to obtain completion information unless the node declaration queue element is the only token

associated with the resource. If this is the case, the task can make a resource request and obtain the token when it is released by XIO on completion.

7.2.3.2 Completion Status

Before processing the completion of the queue element, XIO stores a completion code in the `status` field of the node declaration header, as follows:

0 = Good completion
1 = The node number is out of range or already declared
2 = A queue create system call failed (the queue ID is out of range or the queue already exists)

7.2.4 Host Request Queue Element

When your ICP-resident application or utility task posts a read or write request to the host processor, it must create a queue element and post it to the appropriate node's read or write queue. The queue element consists of two buffers, a header buffer and a data buffer. The header buffer contains a system buffer header followed by a host request header. The next buffer (`sp_nxtb`) field of the system buffer header in the header buffer contains the address of the data buffer. The data buffer also contains a system buffer header, followed by an ICP header, a protocol header, and the received data, if any, that will ultimately be transferred to the application program (in the case of a write request), or the area to which data being sent from the application program will be transferred (in the case of a read request).

Figure 7–4 shows an example of a host request queue element with an encapsulated data buffer.

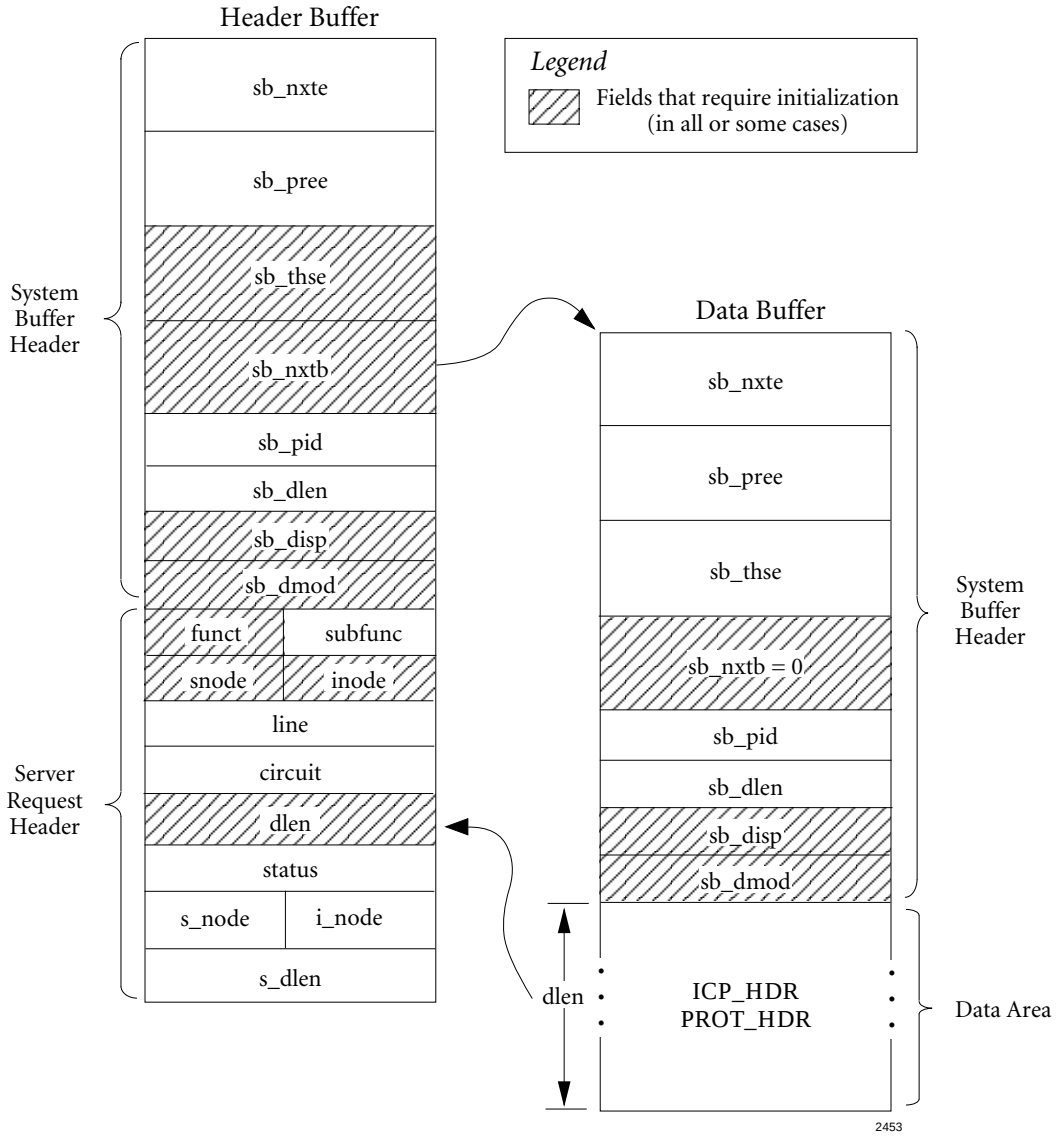


Figure 7–4: Host Request Queue Element with Data Area

The header buffer has the following structure:

```
struct SREQ_HDR_TYPE
{
   struct SBH_TYPE  sbh;
   struct sreq_type req;
};
```

The two structures that make it up are as follows:

```
struct SBH_TYPE
{
       struct SBH_TYPE   *sb_nxte;    /* next element         */
       struct SBH_TYPE   *sb_pree;    /* previous element     */
       struct SBH_TYPE   *sb_thse;    /* this element         */
       struct SBH_TYPE   *sb_nxtb;    /* next buffer          */
       unsigned short    sb_pid;      /* partition ID         */
       unsigned short    sb_dlen;     /* data length          */
       unsigned short    sb_disp;     /* disposition flag     */
       unsigned short    sb_dmod;     /* disposition modifier */
};

struct sreq_type
{
       unsigned char     funct;       /* function code (read or write) */
       unsigned char     subfunct;    /* subfunction code              */
       unsigned char     snode;       /* host node number              */
       unsigned char     inode;       /* ICP node number               */
       unsigned short    line;        /* line number                   */
       unsigned short    circuit;     /* circuit number                */
       unsigned short    dlen;        /* data length, in bytes         */
       unsigned short    status;      /* completion code               */
       unsigned char     s_node;      /* actual host node number
                                         (on completion)               */
       unsigned char     i_node;      /* actual ICP node number
                                         (on completion)               */
       unsigned short    s_dlen;      /* actual number of bytes
                                         transferred                   */
};
```

The data buffer has the following structure:

```
struct data_buffer
{
    struct  SBH_TYPE    sbh;    /* (defined in oscif.h) */
    ICP_HDR  icp_hdr;
    union
    {
        PROT_HDR  prot_hdr;
        XMT_HDR xmt_hdr;
    } prot_hdrs;
    bit8    data;               /* start of data */
};
typedef struct data_buffer DATA_BUFFER;
```

The structures that make it up are as follows:

```
struct SBH_TYPE
{
   struct SBH_TYPE *sb_nxte;   /* next element */
   struct SBH_TYPE *sb_pree;   /* previous element */
   struct SBH_TYPE *sb_thse;   /* this element */
   struct SBH_TYPE *sb_nxtb;   /* next buffer */
   unsigned short   sb_pid;    /* partition ID */
   unsigned short   sb_dlen;   /* data length */
   unsigned short   sb_disp;   /* disposition flag */
   unsigned short   sb_dmod;   /* disposition modifier */
};

struct icp_hdr                      /* ICP message header */
{
        bit16   su_id;              /* service user (client) ID */
        bit16   sp_id;              /* service provider (server) ID */
        bit16   count;              /* size of data following this header */
        bit16   command;            /* function code */
        bit16   status;             /* function status */
        bit16   params[3];          /* API specific parameters  */
};
typedef struct icp_hdr ICP_HDR;
```

```
struct prot_hdr                         /* Protocol message header */
{
        bit16   command;                /* function code */
        bit16   modifier;               /* function modifier */
        bit16   link;                   /* physical port number */
        bit16   circuit;                /* data link circuit identifier */
        bit16   session;                /* session identifier */
        bit16   sequence;               /* message sequence number */
        bit16   reserved1;              /* reserved */
        bit16   reserved2;              /* reserved */
};
typedef struct prot_hdr PROT_HDR;

struct xmt_hdr
{
    bit32   flags;      /* local transmit/receive flags */
    bit8    filler;
    bit8    syncs[8];   /* starting sync chars (BSC)    */
    bit8    start_char; /* start char (BSC)             */
    bit16   count;
};
typedef struct xmt_hdr XMT_HDR;
```

7.2.4.1 System Buffer Header Initialization

In the system buffer header of the header buffer (the first buffer of the queue element), `sb_thse` must be initialized. (This field is set by the system if the buffer was obtained from a partition.) The `sb_nxtb` field must be set to the starting address of the data buffer (that is, to the start of its system buffer header). In addition, the disposition flag, `sb_disp`, and possibly the disposition modifier, `sb_dmod`, must be initialized.

In the system buffer header of the second buffer (the data buffer), initialization of `sb_thse` is not required. If the `sb_nxtb` field is set to zero, the remainder of the buffer immediately follows the system buffer header. If the `sb_nxtb` field is non-zero, it must contain a pointer to the first byte of the API header. For the data buffer, initialization of the system buffer header's disposition flag and disposition modifier might be required, depending on the value of the header buffer's disposition flag.

In the header buffer, the disposition flag must be set to one of the values defined for the field in Section 7.2.1 on page 94. If it is set to POST_QE, FREE_QE, or TOKEN_QE, the disposition flag in the data buffer is ignored. If the disposition flag in the header buffer is set to POST_BUF, FREE_BUF, TOKEN_BUF, or REL_BUF, the disposition flag in the data buffer must also be set to one of those four values, although not necessarily the same one. These options are described in the following paragraphs.

In general, a task requires notification of the completion of a read request so that it can process the message received from the ICP's host. However, it might or might not require notification of the completion of a write request. If the task obtained the buffers of a queue element from a partition, and if it does not require notification when the request has been processed by XIO, the value REL_BUF in the disposition flags of both buffers causes XIO to release the buffers to their partitions on completion.

If the task is maintaining host request queue elements as resource tokens and does not require notification when the request has been processed by XIO, the value TOKEN_QE in the disposition flag and a resource ID in the disposition modifier of the header buffer cause XIO to release the queue element to the resource on completion. Alternatively, the task could maintain the individual buffers of the queue element as resource tokens, in which case TOKEN_BUF should be stored in the disposition flag and resource ID in the disposition modifier of both the header and data buffers.

For notification of the completion of a host request, a task can set the disposition flag in the header buffer to POST_QE, in which case XIO, on completion, posts the queue element, intact, to the queue specified in the disposition modifier of the header buffer. Alternatively, the task can set the disposition flag in the header buffer to FREE_QE, in which case XIO clears the disposition modifier in the header buffer on completion.

If the completion is to be processed separately for the two buffers of the queue element, the requesting task can use the POST_BUF, FREE_BUF, and REL_BUF values, in any combination, for the disposition flags. For example, if the task obtained its data buffer from a partition, but defined a fixed data structure as the header buffer, it might set the dispo-

sition flag in the header buffer to `FREE_BUF` and the disposition flag in the data buffer to `REL_BUF`. Then, when the request is complete, the header buffer is marked free by XIO, indicating to the task that it is available for re-use. The data buffer is released to its partition by XIO, and requires no further processing.

If the disposition flag in either buffer is set to `POST_QE` or `POST_BUF`, the corresponding disposition modifier must contain a valid queue ID. If the requesting task is the owner of the queue, it can suspend its operation and resume when XIO posts the queue element or buffer (with a post and resume, `s_post`, system call) to the queue on completion of the request.

If `FREE_QE` or `FREE_BUF` is specified in the disposition flag of either buffer, the corresponding disposition modifier should be set to a non-zero value so that the requesting task can recognize the completion when the field is cleared by XIO.

7.2.4.2 Host Request Header Initialization

The subfunction, line number, and circuit number fields of the host request header are defined for compatibility with other Protogate products and are not used for the ICP.

The function code must be set to one of the following values:

0x02 = Write request
0x08 = Read request

When a node is declared, a host read request queue ID and a host write request queue ID are defined. For both the main and priority nodes, at least one host request queue element containing a read function code (in the `funct` field of the host request header) must always be posted to that node's read request queue, and a queue element containing a write function code must always be posted to that node's write request queue.

For nodes specific to your ICP-resident task, at least one host request queue element must always be posted to each node's write request queue. For compatibility with other

Protogate implementations, provisions exist for read request queues for these nodes; however, they are not used in the Freeway implementation. In addition, for any node, a host request queue element posted to either the read or write queue must contain a matching ICP node number in the `inode` field of the host request header. The `snode` field should be set as defined for the particular application. (This field is passed to the host, but is not interpreted by XIO. In general, it is used on a write request to specify the destination of the data, and is not used on a read request.)

As the data transfer address for the request, XIO passes to the host the address that is stored in the `sb_nxtb` field of the data buffer's system buffer header. If this value is zero, XIO uses the beginning address of the data buffer plus the length of the system buffer header instead. (The system buffer header itself, as well as any portion of the buffer that separates the system buffer header from the data, are never transferred to or from the host.) For a write request, the `dlen` field of the host request header should be set to the actual number of bytes of data in the data buffer, excluding the system buffer header and any portion of the buffer that separates the header from the data. For a read request, the `dlen` field should be set to the maximum length of the data buffer, excluding the system buffer header and any portion of the buffer that separates the header from the data.

No other fields of the host request header require initialization.

7.2.4.3 Completion Status

Before processing the completion of the queue element, XIO stores a completion code in the `status` field of the host request header, as follows:

If the completion status is good, XIO also returns the node numbers supplied by the host and the actual number of bytes transferred. The ICP node number is returned in the `i_node` field, and always matches the ICP node number supplied by the requesting SPS task in the `inode` field. The host node number is returned in the `s_node` field. For a write, this value always matches the node number supplied by the requesting SPS task

0 = Good completion

1 = The queue to which the host request queue element was posted is defined for a node number other than the one specified in the `inode` field of the host request header

3 = The host request queue element was posted to a host read request queue but contains a write function code, or was posted to a host write request queue but contains a read function code

in the `snode` field. For a read, the node number is generally not specified on request (`snode` is not used), and on completion, the `s_node` field identifies the node number from which the data was received.

Note that `inode` and `i_node` are two separate fields in the host request header, as are `snode` and `s_node`.

The number of bytes actually transferred to or from the host is returned in the `s_dlen` field. This value is never greater than the number requested (`dlen`), but might be less, depending on the data length requested by the corresponding application program.

7.3 Reserved System Resources: XIO Interface

XIO reserves the following system resources:

Queue IDs	1 and 2 (ID 1 = node declaration queue)
Vector numbers	25 and 26 (hexadecimal offsets 64 and 68)
GST entries	`gs_unused` [0] (task entry point) `gs_unused` [1] (panic code)

For proper operation of XIO, ICP-resident SPS tasks added to the system must not use conflicting system resources.

7.4 Executive Input/Output

Executive Input/Output (XIO) consists of three functions which are described in the following sections: `s_initxio`, `s_nodec` and `s_xio`.

The s_initxio function is called once to initialize the internal data structures and devices that allow XIO to communicate with the host's ICP driver. After initialization, the user application can call s_nodec to declare a node. After nodes are declared, the user application issues read and write request using s_xio.

7.4.1 Node Declaration (s_nodec)

The Node Declaration function declares the nodes as described in Section 7.2.3 on page 97. Any error information is returned in the status field of the node declaration header (NODEC_TYPE).

C Interface:

```
s_nodec ( nodec )
struct NODEC_TYPE *nodec;
```

Return: n/a

Assembly Interface:

```
TRAP   #4
```

Input: AO.L = address of NODEC_TYPE structure

Output: none

Access: task or ISR

7.4.2 XIO Read/Write (s_xio)

Issue a read or write request. Depending on the value of the funct field of the host request header (Figure 7–4 on page 101), the s_xio function issues a read or write request.

C Interface:

```
s_xio ( p_hdr )
SREQ_HDR_TYPE *p_hdr;
```

`p_hdr`: pointer to host request header

Assembly Interface:

```
TRAP   #4
```

Input: `A0.L` = address of `SREQ_HDR_TYPE` structure
Output: none

Access:

task or ISR

7.5 Diagnostics

OS/Protogate defines a global system table (GST) that can be accessed at a fixed offset from the load address and contains information used for system initialization and diagnostic purposes. A number of four-byte entries are defined by the operating system as unused and are available for use by ICP-resident system and SPS tasks. (See the *OS/Protogate Operating System Programmer's Guide* for a definition of the GST.)

OS/Protogate initializes the second unused entry in the GST to zero. If OS/Protogate encounters a fatal error during its operation, it stores a panic code at this location and executes an illegal instruction, which causes a trap to the debugger. The panic codes, described below, are each composed of an identifier in the high-order word and a modifier in the low-order word.

Identifier	0x100
Modifier	Error code returned from `s_qcreat`
Description	Creation of the node declaration queue failed (queue ID 1).

Identifier	0x200
Modifier	Error code returned from `s_qcreat`
Description	Creation of the pending request queue failed (queue ID 2).

Identifier	0x300
Modifier	Error code returned from `s_accpt`
Description	An accept message system call on the node declaration queue returned an invalid queue ID error.

Identifier	0x400
Modifier	Queue ID
Description	An accept message system call on a read request queue returned an invalid queue ID error.

Identifier	0x500
Modifier	Queue ID
Description	An accept message system call on a write request queue returned an invalid queue ID error.

Identifier	0x600
Modifier	Error code returned from `s_susp`
Description	A suspend call failed.

Identifier	0x700
Modifier	Disposition flag value
Description	A buffer contains an illegal disposition flag value.

Identifier	0x800
Modifier	Error code returned from `s_accpt`
Description	An accept message system call on the pending request queue returned an invalid queue ID or queue empty error. (The queue should not be empty, because `s_accpt` is not called unless the queue head pointer is non-zero.)

Identifier	0x900
Modifier	8 (error code returned from DMA subroutine)
Description	A data transfer from the host to the ICP failed due to a bus error.

Identifier	0xA00
Modifier	8 (error code returned from DMA subroutine)
Description	A data transfer from the ICP to the host failed due to a bus error.

Chapter

8 Client Applications

This chapter describes how to use the data link interface (DLI) functions, part of Protogate's application program interface (API), to initiate and terminate sessions when developing applications that interface to the ICP sample protocol software (SPS). You should be familiar with the concepts described in the *Freeway Data Link Interface Reference Guide*; however, some summary information is provided in Section 8.1.

The following might be helpful references while reading this chapter:

- Section 8.2 compares a typical sequence of DLI function calls using blocking versus non-blocking I/O.
- Appendix C explains error handling and provides a summary table for error codes. The *Freeway Data Link Interface Reference Guide* gives complete DLI error code descriptions.
- The *Freeway Data Link Interface Reference Guide* provides a generic code example which can guide your application program development, along with the programs described in Appendix D of this manual.

8.1 Summary of DLI Concepts

The DLI presents a consistent, high-level, common interface across multiple clients, operating systems, and transport services. It implements functions that permit your application to use data link services to access, configure, establish and terminate sessions, and transfer data across multiple data link protocols. The DLI concepts are

described in detail in the *Freeway Data Link Interface Reference Guide*. This section summarizes the basic information.

8.1.1 Configuration in the Freeway Server or Embedded Environment

Several items must be configured before a client application can run in the Freeway environment:

- boot configuration for Freeway server implementations
- data link interface (DLI) session configuration
- transport subsystem interface (TSI) connection configuration
- protocol-specific ICP link configuration

The Freeway server boot configuration file is normally created during the installation procedures described in the *Freeway Server User's 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. The DLI and TSI configuration process is described in Section 8.1.1.1 and Section 8.1.1.2.

Protocol-specific ICP link configuration must be performed by the client application (as described in Section 8.5.7.1 on page 142 and Section 9.1.1 on page 152) after `dlOpen` completes the DLI session establishment process.

8.1.1.1 DLI Configuration for *Raw* Operation

The application program interface (API) is implemented in two levels: the data link interface (DLI) and the transport subsystem interface (TSI). These levels are documented in the *Freeway Data Link Interface Reference Guide* and the *Freeway Transport Subsystem Interface Reference Guide*.

The DLI provides two levels of operation for ICP protocol software, as described in the *Freeway Data Link Interface Reference Guide*. *Normal* operation is not supported by the SPS. *Raw* operation means that the application programmer must provide link configuration, link enable, and all the other requirements of the ICP protocol software. The SPS is provided as an example to be modified; however, the DLI supports only *Raw* operation for the SPS. The DLI optional arguments data structure (`DLI_OPT_ARGS`), which is central to *Raw* operation, is described in Section 8.4 on page 126.

The configuration files for the client application are relatively simple. However, you must specify the DLI configuration parameters whose values differ from the defaults. Figure 8–1 shows a portion of a typical DLI configuration file, such as `spsaldcfg`. The `BoardNo` parameter specifies the target ICP. If `BoardNo` is not specified, the default is zero. The `PortNo` parameter may or may not be provided. The `PortNo` parameter is required if the application requests a DLI session status and expects to see the correct value for `iPortNo`. If your application does not require the `iPortNo` value, the DLI configuration file does not need to specify `PortNo`. If `PortNo` is not included, only one DLI section (besides the "main" section) is required in the configuration file, which can be referenced in all calls to `dlOpen`. Refer to the *Freeway Data Link Interface Reference Guide* for more information on requesting DLI session status.

8.1.1.2 DLI and TSI Configuration Process

This section summarizes the process for configuring DLI sessions and TSI connections. DLI and TSI text configuration files are used as input to the `dlicfg` and `tsicfg` preprocessor programs to produce binary configuration files which are used by the `dlInit` and `dlOpen` functions. The DLI and TSI configuration process is a part of the loopback testing procedure described in Appendix D and the installation procedure described in the *Freeway Server User's Guide*. However, during your client application development and testing, you might need to perform DLI and TSI configuration repeatedly. These procedures are summarized as follows:

```
//--------------------------------------------------------------------//
// "main" section.  If not defined defaults are used. If present   //
// the main section must be the very first section of the DLI      //
// configuration file.                                              //
//--------------------------------------------------------------------//

main
{
    AsyncIO = "yes";                // Non-blocking I/O             //
    TSICfgName = "spsaltcfg.bin";   // TSI binary config file       //
}

//--------------------------------------------------------------------//
// Define a section for a raw port.                                 //
//--------------------------------------------------------------------//

serverOicpOport0
{
    AlwaysQIO = "yes";               // DLI always queues I/O       //
    AsyncIO = "Yes";                 // Non-blocking I/O            //
    BoardNo = 0;                     // First ICP is board 0        //
    CfgLink = "No";                  // Client must configure link  //
    Enable = "No";                   // Client must enable link     //
    PortNo = 0;                      // First link is 0             //
    Protocol = "raw";                // SPS uses Raw operation      //
    Transport = "conn0";             // TSI connection name         //
}
//--------------------------------------------------------------------//
// Define a section for a raw port.                                 //
//--------------------------------------------------------------------//

serverOicpOport1
{
    AlwaysQIO = "yes";               // DLI always queues I/O       //
    AsyncIO = "Yes";                 // Non-blocking I/O            //
    BoardNo = 0;                     // First ICP is board 0        //
    CfgLink = "No";                  // Client must configure link  //
    Enable = "No";                   // Client must enable link     //
    PortNo = 1;                      // Second link is 1            //
    Protocol = "raw";                // SPS uses Raw operation      //
    Transport = "conn0";             // TSI connection name         //
}
```

Figure 8–1: Typical DLI "main" Configuration plus Two Sessions

1. Create or modify a TSI text configuration file specifying the configuration of the TSI connections (for example, `spsaltcfg` in the `freeway/client/test/sps` directory).

2. Create or modify a DLI text configuration file specifying the DLI session configuration for all ICPs and serial communication links in your system (for example, `spsaldcfg` in the `freeway/client/test/sps` directory).

3. If you have a UNIX or Windows NT system, skip this step. If you have a VMS system, run the `makefc.com` command file from the `[FREEWAY.CLIENT.TEST.SPS]` directory to create the foreign commands used for `dlicfg` and `tsicfg`.

   ```
   @MAKEFC UCX
   ```

4. From the `freeway/client/test/sps` directory, execute `tsicfg` with the text file from Step 1 as input. This creates the TSI binary configuration file in the same directory as the location of the text file (unless a different path is supplied with the optional filename). If the optional filename is not supplied, the binary file is given the same name as your TSI text configuration file plus a `.bin` extension.

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

UNIX example:	`freeway/client/`*op-sys*`/bin/tsicfg spsaltcfg`
VMS example:	`tsicfg spsaltcfg`
Windows NT example:	`freeway\client\`*op-sys*`\bin\tsicfg spsaltcfg`

5. From the `freeway/client/test/sps` directory, execute `dlicfg` with the text file from Step 2 as input. This creates the DLI binary configuration file in the same directory as the location of the text file (unless a different path is supplied with the optional filename). If the optional filename is not supplied, the binary file is given the same name as your DLI text configuration file plus a `.bin` extension.

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

UNIX example:	`freeway/client/`*op-sys*`/bin/dlicfg spsaldcfg`

VMS example: `dlicfg spsaldcfg`

Windows NT example: `freeway\client\`*op-sys*`\bin\dlicfg spsaldcfg`

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. On all but VMS systems, if a binary file already exists with the same name in the directory, the existing file is renamed by appending the `.BAK` extension. If the renamed file duplicates an existing file in the directory, the existing file is removed by the configuration preprocessor program.

6. If you have a UNIX system, move the TSI and DLI binary configuration files that you created in Step 4 and Step 5 into the appropriate `freeway/client/`*op-sys*`/bin` directory where *op-sys* indicates the operating system: `hpux`, `solaris`, `sunos`, or `osf1`. For example, `freeway/client/sunos/bin`.

7. If you have a VMS system, run the `move.com` command file from the `[FREEWAY.CLIENT.TEST.SPS]` directory. This moves the DLI and TSI binary configuration files you created in Step 4 and Step 5 into the `bin` directory for your particular TCP/IP package.

   `@MOVE` *filename* `UCX`

8. If you have a Windows NT system, move the TSI and DLI binary configuration files that you created in Step 4 and Step 5 into the appropriate `freeway\client\`*op-sys*`\bin` directory where *op-sys* indicates the operating system: `ant` or `int` for a Freeway server or `int_nt_emb` for an embedded system. For example, `freeway\client\ant\bin`.

When your application calls the `dlInit` function, the DLI and TSI binary configuration files generated in Step 4 and Step 5 are used to configure the DLI sessions and TSI con-

nections. The *Freeway Transport Subsystem Interface Reference Guide* provides additional details on the TSI configuration. Figure 8–2 shows the configuration process.

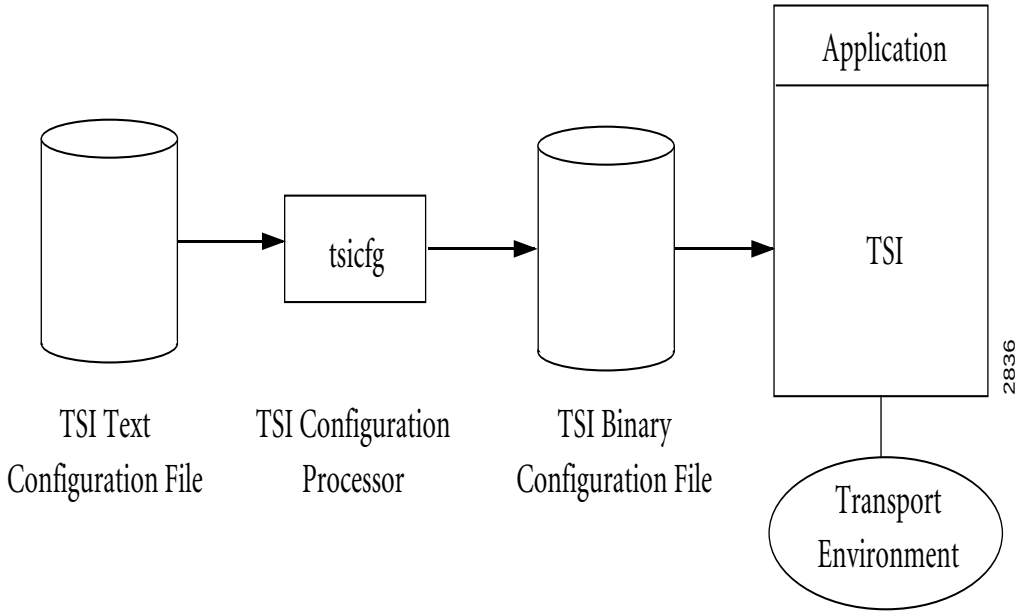


Figure 8–2: DLI and TSI Configuration Process

8.1.2 Blocking versus Non-blocking I/O

Note

Earlier Protogate 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 waiting for

I/O completion. Blocking I/O applications are useful when it is reasonable to have the calling process wait for I/O completion.

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 ICP 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 determine the I/O completion status.

The `asyncIO` DLI configuration parameter specifies whether an application session uses blocking or non-blocking I/O (set `asyncIO` to "no" to use blocking I/O). The `alwaysQIO` DLI configuration parameter further qualifies the operation of non-blocking I/O activity. Refer to the *Freeway Data Link Interface Reference Guide* for more information.

The effects on different DLI functions, resulting from the choice of blocking or non-blocking I/O, are explained in the *Freeway Data Link Interface Reference Guide*.

Server-resident applications must use non-blocking I/O; support for blocking I/O in server-resident applications is not available.

8.1.3 Buffer Management

Currently the interrelated Freeway server, DLI, TSI, and ICP buffers default to a size of 1024 bytes.

Caution

If you need to change a buffer size for your application, refer to the *Freeway Data Link Interface Reference Guide* for explanations of the complexities that you must consider.

8.2 Example Call Sequences

Table 8–1 shows the sequence of DLI function calls to send and receive data using blocking I/O. Table 8–2 is the non-blocking I/O example. The remainder of this chapter and the *Freeway Data Link Interface Reference Guide* give further information about each function call. Refer back to Section 8.1.2 on page 119 for more information on blocking and non-blocking I/O.

Note

The example call sequences assume that the `cfgLink` and `enable` DLI configuration parameters are set to "no" (the default is "yes" for both). This is necessary for the client application to configure and enable the ICP links. Figure 8–1 on page 116 shows an example DLI configuration file.

Table 8–1: DLI Call Sequence[a] for Blocking I/O

1. Call `dlInit` to initialize the DLI operating environment. The first parameter is your DLI binary configuration file name.
2. Call `dlOpen` for each required session (link) to get a session ID.
3. Call `dlBufAlloc` for all required input and output buffers.
4. Call `dlWrite` to send an attach request to Freeway.
5. Call `dlRead` to receive the protocol session ID from Freeway.
6. Call `dlWrite` to send a configuration message to Freeway.
7. Call `dlRead` to receive the configuration confirmation from Freeway.
8. Call `dlWrite` to send a link activation message to Freeway.
9. Call `dlRead` to receive the link activation confirmation from Freeway.
10. Call `dlWrite` to send requests and data to Freeway.
11. Call `dlRead` to receive responses and data from Freeway.
12. Repeat Step 10 and Step 11 until you are finished writing and reading.
13. Call `dlBufFree` for all buffers allocated in Step 3.
14. Call `dlClose` for each session ID obtained in Step 2.
15. Call `dlTerm` to terminate your application's access to Freeway.

[a] Because the protocol software can send a message to the client at any time, a `dlRead` request must always be queued to avoid loss of data or response messages from the ICP.

Table 8–2: DLI Call Sequence[a] for Non-blocking I/O

1. Call `dlInit` to initialize the DLI operating environment. The first parameter is your DLI binary configuration file name.
2. Call `dlOpen` for each required session (link) to get a session ID.
3. Call `dlPoll` to confirm the success of each session ID obtained in Step 2.
4. Call `dlBufAlloc` for all required input and output buffers.
5. Call `dlWrite` to send an attach request to Freeway.
6. Call `dlRead` to receive the protocol session ID from Freeway.
7. Call `dlWrite` to send a configuration message to Freeway.
8. Call `dlRead` to receive the configuration confirmation from Freeway.
9. Call `dlWrite` to send a link activation message to Freeway.
10. Call `dlRead` to receive the link activation confirmation from Freeway.
11. Call `dlWrite` to send requests and data to Freeway.
12. Call `dlRead` to queue reads to receive responses and data from Freeway.
13. As I/Os complete, call `dlPoll` to confirm the success of each `dlWrite` in Step 11 and to accept the data from each `dlRead` in Step 12.
14. Repeat Step 11 through Step 13 until you are finished writing and reading.
15. Call `dlBufFree` for all buffers allocated in Step 4.
16. Call `dlClose` for each session ID obtained in Step 2.
17. Call `dlPoll` to confirm that each session was closed in Step 16.
18. Call `dlTerm` to terminate your application's access to Freeway.

[a] Because the protocol software can send a message to the client at any time, a `dlRead` request must always be queued to avoid loss of data or response messages from the ICP.

Note

Server-resident applications must use non-blocking I/O. It is also necessary to call `dlPost` before relinquishing task control. See the *Freeway Data Link Interface Reference Guide* for details.

8.3 Overview of DLI Functions

After the protocol software is downloaded to the ICP, the client and ICP 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.

Caution

A `dlRead` request must always be queued to avoid loss of data or responses from the ICP.

This section summarizes the DLI functions used in writing a client application. The simplest view of using the DLI functions is:

- Start up communications (`dlInit`, `dlOpen`, `dlBufAlloc`, `dlWrite`, `dlRead`)
- Send requests and data using `dlWrite`
- Receive responses using `dlRead`
- 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
- For server-resident applications, use `dlPost` before relinquishing task control
- Shut down communications (`dlBufFree`, `dlClose`, `dlTerm`)

Table 8–3 summarizes the DLI function syntax and parameters, listed in the most likely calling order. Refer to the *Freeway Data Link Interface Reference Guide* for details.

The remainder of this chapter and Chapter 9 describe the `dlWrite` and `dlRead` functions. Both functions use the optional arguments parameter to provide the protocol-specific information required for *Raw* operation (see Section 8.1.1.1 on page 114). The "C" definition of the optional arguments is described in Section 8.4 on page 126.

Table 8–3: DLI Functions: Syntax and Parameters (Listed in Typical Call Order)

DLI Function	Parameter(s)	Parameter Usage

`int dlInit`	`(char *cfgFile,`
	`char *pUsrCb,`
	`int (*fUsrIOCH)(char *pUsrCb));`
		DLI binary configuration file name
		Optional I/O complete control block
		Optional IOCH and parameter

`int dlOpen`[a]	`(char *cSessionName,`
	`int (*fUsrIOCH)`
	`(char *pUsrCB, int iSessionID));`
		Session name in DLI config file
		Optional I/O completion handler
		Parameters for IOCH

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

`char *dlBufAlloc`	`(int iBufLen);`	Minimum buffer size

`int dlRead`	`(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`	`(int iSessionID,`
	`char *pBuf,`
	`int iBufLen,`
	`int iWritePriority,`
	`DLI_OPT_ARGS *pOptArgs);`
		Session ID from `dlOpen`
		Source buffer for write
		Number of bytes to write
		Normal or expedite write
		Optional arguments structure

`int dlPost`	`(void);`	
`char *dlBufFree`	`(char *pBuf);`	Buffer to return to pool
`int dlClose`	`(int iSessionID,` `int iCloseMode);`	Session ID from `dlOpen` Mode (normal or force)
`int dlTerm`	`(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

[a] 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` requests.

8.4 Client and ICP Interface Data Structures

The data link interface (DLI) provides a session-level interface between a client application and the sample protocol software resident on an ICP. Messages traveling from the client application go over the Ethernet to the Freeway server or ICP driver and end up at the ICP. From the client's perspective, these messages consist of data buffers supplemented with the DLI optional arguments data structure to provide the protocol-specific information required for *Raw* operation (Section 8.1.1.1 on page 114). Figure 8–3 shows the "C" definition of the DLI optional arguments structure.

```
typedef struct {
unsigned short   usFWPacketType;     /* Client's packet type */
unsigned short   usFWCommand;        /* Client's cmd sent or rcvd */
unsigned short   usFWModifier;       /* Client's cmd modifier */
unsigned short   usFWStatus;         /* Client'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 xtra parameters */
unsigned short   usProtCommand;      /* protocol cmd */
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 xtra parms */
} DLI_OPT_ARGS;
```

Figure 8–3: "C" Definition of DLI Optional Arguments Structure

From the ICP's perspective, these messages consist of the api_msg data structure shown in Figure 8–4.

```
struct api_msg {
       ICP_HDR  icp_hdr;
       PROT_HDR prot_hdr;
       bit8     *data;
};
```

Figure 8–4: "C" Definition of api_msg Data Structure

The icp_hdr structure is of type ICP_HDR and the prot_hdr structure is of type PROT_HDR, as shown in Figure 8–5.

```
typedef struct {              /* ICP message header                 */
       bit16   su_id;         /* service user (client) ID           */
       bit16   sp_id;         /* service provider (server) ID       */
       bit16   count;         /* size of data following this header */
       bit16   command;       /* function code                      */
       bit16   status;        /* function status                    */
       bit16   params[3];     /* ICP-specific parameters            */
} ICP_HDR;

typedef struct {              /* Protocol message header            */
       bit16   command;       /* function code                      */
       bit16   modifier;      /* function modifier                  */
       bit16   link;          /* physical port number               */
       bit16   circuit;       /* data link circuit identifier       */
       bit16   session;       /* session identifier                 */
       bit16   sequence;      /* message sequence number            */
       bit16   reserved1;     /* reserved                           */
       bit16   reserved2;     /* reserved                           */
} PROT_HDR;
```

Figure 8–5: "C" Definition of icp_hdr and prot_hdr Data Structures

Table 8–4 shows the equivalent fields between the DLI_OPT_ARGS structure and the ICP_HDR and PROT_HDR structures.

Table 8–4: Equivalent Fields between DLI_OPT_ARGS and ICP_HDR/PROT_HDR

DLI_OPT_ARGS in DLI Client Program	ICP_HDR and PROT_HDR in ICP SPS Program	Field Description
DLI_OPT_ARGS.usFWPacketType	unused	client's packet type
DLI_OPT_ARGS.usFWCommand	unused	client's command sent or received
DLI_OPT_ARGS.usFWStatus	unused	client's status of I/O operations
DLI_OPT_ARGS.usICPClientID	icp.su_id	old su_id
DLI_OPT_ARGS.usICPServerID	icp.sp_id	old sp_id
count filled in by DLI	icp.count	data size
DLI_OPT_ARGS.usICPCommand	icp.command	ICP's command
DLI_OPT_ARGS.iICPStatus	icp.status	ICP's command status
DLI_OPT_ARGS.usICPParms[3]	icp.params[3]	ICP's extra parameters
DLI_OPT_ARGS.usProtCommand	prot.command	protocol command
DLI_OPT_ARGS.iProtModifier	prot.modifier	protocol command's modifier
DLI_OPT_ARGS.usProtLinkID	prot.link	protocol link ID
DLI_OPT_ARGS.usProtCircuitID	prot.circuit	protocol circuit ID
DLI_OPT_ARGS.usProtSessionID	prot.session	protocol session ID
DLI_OPT_ARGS.usProtSequence	prot.sequence	protocol sequence
DLI_OPT_ARGS.usProtXParms[2]	prot.reserved1	protocol extra parameters
	prot.reserved2	second XParms field

The client API translates between the DLI_OPT_ARGS and the api_msg data structures. The usICPCommand field of the DLI_OPT_ARGS structure corresponds to the command field of the ICP_HDR structure. The usProtCommand field of the DLI_OPT_ARGS structure corresponds to the command field of the PROT_HDR structure.

The ICP supports the following commands:

- attach
- bind
- write
- unbind
- detach

These commands are encoded into the `DLI_OPT_ARGS` structure's `usICPCommand` field. In addition, for the `write` command, the `usProtCommand` can specify the following qualifiers:

- configure link
- provide statistics
- transmit data

The following sections describe how these commands are used to access and provide data to a wide area network.

8.5 Client and ICP Communication

The following sections discuss the DLI functions and `DLI_OPT_ARGS` data structure as used by client applications in communicating with ICP software. In addition, this communication is discussed from the ICP perspective with details regarding the content of the `ICP_HDR` and `PROT_HDR` data structures.

8.5.1 Sequence of Client Events to Communicate to the ICP

To exchange data with a wide-area network, a client must follow these steps:

1. Initiate a session with the Freeway server or the embedded product's driver
2. Initiate a session with the ICP link
3. Configure the link
4. Activate the link
5. Send data to and receive data from the link
6. Deactivate the link
7. End the session with the ICP link
8. End the session with the Freeway server or the embedded product's driver

The following sections describe how to use the DLI subroutine library to perform these steps. Prior to these steps, however, the DLI must be initialized. This is accomplished when the application calls the `dlInit` function, which is declared as follows:

```
int dlInit (char *pCfgFile,
            char *pUsrCB,
            int  (*pUsrIOCH) (char *pUsrCB));
```

The following is an example of a call to `dlInit`:

```
status = dlInit ("spsaldcfg.bin", NULL, NULL);
```

This example indicates to DLI that the file `spsaldcfg.bin` is available to be read to configure the process, and that if an I/O completion function is to be called, it is specified as individual sessions are opened in calls to `dlOpen`. For more information, consult the *Freeway Data Link Interface Reference Guide*.

8.5.2 Initiating a Session with the ICP

A session identifier is used by DLI to manage information exchanged between the client application and the ICP. The session identifier is requested by the client, then defined and returned by the DLI. This is accomplished when the application calls `dlOpen`. The ICP software is not involved in these two steps. The `dlOpen` function is declared as follows:

```
int dlOpen (char   *cSessionName,
            short  (*fUsrIOCH) (char *pUseCB,
            int  iSessionID));
```

The first argument is the name of a section in the application's DLI configuration file. The second argument is the name of a function, supplied by the application writer, that DLI calls when it services an I/O condition for the session identifier returned by the `dlOpen` call.

The following is an example of a call to `dlOpen`.

```
servSessID = dlOpen ("serverOicpOportO", ioComplete);
```

The string `serverOicpOportO` is the name of a section in a DLI configuration file, and `ioComplete` is the name of a function the application writer provides. The value of `servSessID` is used in further calls to DLI functions.

8.5.3 Initiating a Session with an ICP Link

When the DLI configuration file parameter `protocol` is set to `raw` (`protocol="raw"`), a call to the `dlOpen` function establishes a data path to the ICP for a given link. This path is referenced by the return value of `dlOpen` and is called the session identifier. A call to `dlPoll` can be used to verify the success of the `dlOpen` function. If the status of the new session is `DLI_STATUS_READY`, the open was successful. Next, the data path must be extended to the ICP with an attach. This is accomplished by issuing a call to `dlWrite` with the optional argument structure set as follows:

`DLI_OPT_ARGS.usFWPacketType`	`FW_DATA`
`DLI_OPT_ARGS.usFWCommand`	`FW_ICP_WRITE`
`DLI_OPT_ARGS.usFWStatus`	*n/a*
`DLI_OPT_ARGS.usICPClientID`	*n/a*
`DLI_OPT_ARGS.usICPServerID`	*n/a*
`DLI_OPT_ARGS.usICPCommand`	`DLI_ICP_CMD_ATTACH`
`DLI_OPT_ARGS.ilCPStatus`	*n/a*
`DLI_OPT_ARGS.usICPParms [0]`	*n/a*
`DLI_OPT_ARGS.usICPParms [1]`	*n/a*
`DLI_OPT_ARGS.usICPParms [2]`	*n/a*
`DLI_OPT_ARGS.usProtCommand`	`DLI_ICP_CMD_ATTACH`
`DLI_OPT_ARGS.iProtModifier`	*n/a*
`DLI_OPT_ARGS.usProtLinkID`	*Link the session relates to*
`DLI_OPT_ARGS.usProtCircuitID`	*n/a*
`DLI_OPT_ARGS.usProtSessionID`	*n/a*
`DLI_OPT_ARGS.usProtSequence`	*n/a*
`DLI_OPT_ARGS.usProtXParms [0]`	*n/a*
`DLI_OPT_ARGS.usProtXParms [1]`	*n/a*

The response to the attach is read with a call to `dlRead`. If the attach was successful, the optional argument structure in the response is as follows:

`DLI_OPT_ARGS.usFWPacketType`	*n/a*
`DLI_OPT_ARGS.usFWCommand`	*n/a*
`DLI_OPT_ARGS.usFWStatus`	*n/a*
`DLI_OPT_ARGS.usICPClientID`	*n/a*
`DLI_OPT_ARGS.usICPServerID`	*n/a*
`DLI_OPT_ARGS.usICPCommand`	`DLI_ICP_CMD_ATTACH`
`DLI_OPT_ARGS.ilCPStatus`	`DLI_ICP_ERR_NO_ERR`
`DLI_OPT_ARGS.usICPParms [0]`	*n/a*
`DLI_OPT_ARGS.usICPParms [1]`	*n/a*
`DLI_OPT_ARGS.usICPParms [2]`	*n/a*
`DLI_OPT_ARGS.usProtCommand`	`DLI_ICP_CMD_ATTACH`
`DLI_OPT_ARGS.iProtModifier`	*n/a*
`DLI_OPT_ARGS.usProtLinkID`	*link the session relates to*
`DLI_OPT_ARGS.usProtCircuitID`	*n/a*
`DLI_OPT_ARGS.usProtSessionID`	*Protocol Session ID*
`DLI_OPT_ARGS.usProtSequence`	*n/a*
`DLI_OPT_ARGS.usProtXParms [0]`	*n/a*
`DLI_OPT_ARGS.usProtXParms [1]`	*n/a*

The ICP returns a protocol session identifier in the `usProtSessionID` field. This value must be used in the `usProtSessionID` field of the optional arguments structure in all future calls to `dlWrite` for this link.

From the ICP's perspective, the `attach` command establishes a session between the client application and one of the ICP links. A successful `attach` command gives the ICP and the client application IDs that are unique to the current session with which they can relay information.

The protocol session identifier is used by the ICP to manage information exchanged between the client application and a specific link. The protocol session identifier is requested by the client, then defined and returned by the ICP.

For the attach command, the fields of the ICP and protocol headers that the ICP receives contain the following values:

```
ICP_HDR
count       = size of icp header
command     = DLI_ICP_CMD_ATTACH
status      = high bits indicate byte ordering
params[0]   = return node number

PROT_HDR
link        = link number
```

After the ICP processes the attach command, it returns these headers with the following field modifications:

```
ICP_HDR
status      = error code or zero if successful

PROT_HDR
modifier    = error code or zero if successful
session     = session ID assigned by the ICP
```

The ICP receives an ICP header containing DLI_ICP_CMD_ATTACH in the command field and a return node number (assigned by msgmux for the Freeway server or the ICP driver for the Freeway embedded product) in the params[0] field. It also receives a link number in the link field of the protocol header. If the ICP can successfully complete the attach, it returns a session number in the session field of the protocol header and a zero (indicating success) in both the status field of the ICP header and the modifier field of the protocol header. Any subsequent transactions involving this session number will be transmitted from the ICP via the corresponding node number.

There is a correspondence between node numbers and session numbers. (Chapter 7 provides more information on node numbers and the host/ICP interface). All the commands in Section 8.5.1 on page 130, from the attach command on, must have this session number in the session field of the protocol header. (The msgmux or ICP driver copies the protocol session ID from the usProtSessionID field in the client's DLI_OPT_ARGS to the session field in the protocol header.) If the attach is unsuccessful

(for example, the link has already been attached or the link or node number is invalid), the ICP returns an appropriate error code in the `status` field of the ICP header and the `modifier` field of the protocol header. The *Freeway Data Link Interface Reference Guide* lists possible error codes.

8.5.4 Terminating a Session with an ICP Link

When the DLI configuration file parameter `protocol` is set to `raw` (`protocol="raw"`), a call to the `dlClose` function terminates a data path to the ICP for a given link. However, before the session is terminated, it is important to allow the ICP to release the space allocated by it for session management. This is accomplished by issuing a call to `dlWrite` with the optional argument structure set as follows:

`DLI_OPT_ARGS.usFWPacketType`	`FW_DATA`
`DLI_OPT_ARGS.usFWCommand`	`FW_ICP_WRITE`
`DLI_OPT_ARGS.usFWStatus`	*n/a*
`DLI_OPT_ARGS.usICPClientID`	*n/a*
`DLI_OPT_ARGS.usICPServerID`	*n/a*
`DLI_OPT_ARGS.usICPCommand`	`DLI_ICP_CMD_DETACH`
`DLI_OPT_ARGS.ilCPStatus`	*n/a*
`DLI_OPT_ARGS.usICPParms [0]`	*n/a*
`DLI_OPT_ARGS.usICPParms [1]`	*n/a*
`DLI_OPT_ARGS.usICPParms [2]`	*n/a*
`DLI_OPT_ARGS.usProtCommand`	`DLI_ICP_CMD_DETACH`
`DLI_OPT_ARGS.iProtModifier`	*n/a*
`DLI_OPT_ARGS.usProtLinkID`	*link the session relates to*
`DLI_OPT_ARGS.usProtCircuitID`	*n/a*
`DLI_OPT_ARGS.usProtSessionID`	*Session to end*
`DLI_OPT_ARGS.usProtSequence`	*n/a*
`DLI_OPT_ARGS.usProtXParms [0]`	*n/a*
`DLI_OPT_ARGS.usProtXParms [1]`	*n/a*

The response to the detach is read with a call to `dlRead`. If the detach was successful, the optional argument structure in the response is as follows:

`DLI_OPT_ARGS.usFWPacketType`	*n/a*
`DLI_OPT_ARGS.usFWCommand`	*n/a*
`DLI_OPT_ARGS.usFWStatus`	*n/a*
`DLI_OPT_ARGS.usICPClientID`	*n/a*
`DLI_OPT_ARGS.usICPServerID`	*n/a*
`DLI_OPT_ARGS.usICPCommand`	`DLI_ICP_CMD_DETACH`
`DLI_OPT_ARGS.ilCPStatus`	`DLI_ICP_ERR_NO_ERR`
`DLI_OPT_ARGS.usICPParms [0]`	*n/a*
`DLI_OPT_ARGS.usICPParms [1]`	*n/a*
`DLI_OPT_ARGS.usICPParms [2]`	*n/a*
`DLI_OPT_ARGS.usProtCommand`	`DLI_ICP_CMD_DETACH`
`DLI_OPT_ARGS.iProtModifier`	*n/a*
`DLI_OPT_ARGS.usProtLinkID`	*link the session relates to*
`DLI_OPT_ARGS.usProtCircuitID`	*n/a*
`DLI_OPT_ARGS.usProtSessionID`	*Protocol Session ID*
`DLI_OPT_ARGS.usProtSequence`	*n/a*
`DLI_OPT_ARGS.usProtXParms [0]`	*n/a*
`DLI_OPT_ARGS.usProtXParms [1]`	*n/a*

Once the ICP releases the session, the application can call `dlClose` to terminate the session with the ICP.

From the ICP's perspective, the `detach` command terminates an active session between the client and the ICP. When the application is finished with the ICP session, it writes a `detach` command to the ICP. The application establishes a `DLI_OPT_ARGS` structure requesting the detach, then sends the structure to the ICP with a call to the DLI `dlWrite` function.

For the `detach` command, the fields of the ICP and protocol headers that the ICP receives contain the following values:

```
ICP_HDR
count       = size of icp header
command     = DLI_ICP_CMD_DETACH
status      = high bits indicate byte ordering

PROT_HDR
session     = session ID
```

The ICP receives a message consisting of an ICP header with DLI_ICP_CMD_DETACH in the command field and a protocol header with the session number in the session field. The ICP responds to this command by making that session's ID available for future sessions. The ICP also turns off devices and clears the link control table's link active flag for that session's link if this was not already done as a result of a prior unbind command. The ICP always puts a zero, indicating success, in the status field of the ICP header and the modifier field of the protocol header and sends the two headers back to the client as an acknowledgment.

8.5.5 Activating an ICP Link

Once dlOpen has been called and an attach message written to the ICP, the link can be configured. After the link is configured, it is necessary to request the ICP to start the link's receiver and transmitter. Starting (enabling) the link is accomplished by sending a bind message to the ICP. This is accomplished by issuing a call to dlWrite with the optional argument structure set as follows:

DLI_OPT_ARGS.usFWPacketType	FW_DATA
DLI_OPT_ARGS.usFWCommand	FW_ICP_WRITE
DLI_OPT_ARGS.usFWStatus	*n/a*
DLI_OPT_ARGS.usICPClientID	*n/a*
DLI_OPT_ARGS.usICPServerID	*n/a*
DLI_OPT_ARGS.usICPCommand	DLI_ICP_CMD_BIND
DLI_OPT_ARGS.ilCPStatus	*n/a*
DLI_OPT_ARGS.usICPParms [0]	*n/a*
DLI_OPT_ARGS.usICPParms [1]	*n/a*
DLI_OPT_ARGS.usICPParms [2]	*n/a*
DLI_OPT_ARGS.usProtCommand	DLI_ICP_CMD_BIND
DLI_OPT_ARGS.iProtModifier	*n/a*
DLI_OPT_ARGS.usProtLinkID	*Link to start*
DLI_OPT_ARGS.usProtCircuitID	*n/a*
DLI_OPT_ARGS.usProtSessionID	*Protocol Session ID*
DLI_OPT_ARGS.usProtSequence	*n/a*
DLI_OPT_ARGS.usProtXParms [0]	*n/a*
DLI_OPT_ARGS.usProtXParms [1]	*n/a*

The response to the bind is a read with a call to `dlRead`. If the bind was successful, the optional argument structure in the response is as follows:

`DLI_OPT_ARGS.usFWPacketType`	*n/a*
`DLI_OPT_ARGS.usFWCommand`	*n/a*
`DLI_OPT_ARGS.usFWStatus`	*n/a*
`DLI_OPT_ARGS.usICPClientID`	*n/a*
`DLI_OPT_ARGS.usICPServerID`	*n/a*
`DLI_OPT_ARGS.usICPCommand`	`DLI_ICP_CMD_BIND`
`DLI_OPT_ARGS.ilCPStatus`	`DLI_ICP_ERR_NO_ERR`
`DLI_OPT_ARGS.usICPParms [0]`	*n/a*
`DLI_OPT_ARGS.usICPParms [1]`	*n/a*
`DLI_OPT_ARGS.usICPParms [2]`	*n/a*
`DLI_OPT_ARGS.usProtCommand`	`DLI_ICP_CMD_BIND`
`DLI_OPT_ARGS.iProtModifier`	*n/a*
`DLI_OPT_ARGS.usProtLinkID`	*Link started*
`DLI_OPT_ARGS.usProtCircuitID`	*n/a*
`DLI_OPT_ARGS.usProtSessionID`	*Protocol Session ID*
`DLI_OPT_ARGS.usProtSequence`	*n/a*
`DLI_OPT_ARGS.usProtXParms [0]`	*n/a*
`DLI_OPT_ARGS.usProtXParms [1]`	*n/a*

The `bind` command activates one of the ICP's links by initializing flags and turning on that link's receiver.

From the ICP's perspective, when the application sends a `bind` command to the ICP, the ICP completes all preparations to receive and transmit on the specified link. For the `bind` command, the fields of the ICP and protocol headers that the ICP receives contain the following values:

```
ICP_HDR
count       = size of icp header
command     = DLI_ICP_CMD_BIND
status      = high bits indicate byte ordering

PROT_HDR
session     = session ID
```

The constant value `DLI_ICP_CMD_BIND` is in the `command` field of the ICP header and a session number is in the `session` field of the protocol header. The ICP starts that link's receiver, sets the link control table link active flag, and returns an acknowledgment to

the client. If the link was already active, the acknowledgment contains an error code in the ICP header's `status` field and the protocol header's `modifier` field. Otherwise they contain zero, indicating success.

8.5.6 Deactivating an ICP Link

Stopping (disabling) the link is accomplished by sending an unbind message to the ICP. This is accomplished by issuing a call to `dlWrite` with the optional argument structure set as follows:

`DLI_OPT_ARGS.usFWPacketType`	`FW_DATA`
`DLI_OPT_ARGS.usFWCommand`	`FW_ICP_WRITE`
`DLI_OPT_ARGS.usFWStatus`	*n/a*
`DLI_OPT_ARGS.usICPClientID`	*n/a*
`DLI_OPT_ARGS.usICPServerID`	*n/a*
`DLI_OPT_ARGS.usICPCommand`	`DLI_ICP_CMD_UNBIND`
`DLI_OPT_ARGS.ilCPStatus`	*n/a*
`DLI_OPT_ARGS.usICPParms [0]`	*n/a*
`DLI_OPT_ARGS.usICPParms [1]`	*n/a*
`DLI_OPT_ARGS.usICPParms [2]`	*n/a*
`DLI_OPT_ARGS.usProtCommand`	`DLI_ICP_CMD_UNBIND`
`DLI_OPT_ARGS.iProtModifier`	*n/a*
`DLI_OPT_ARGS.usProtLinkID`	*Link to stop*
`DLI_OPT_ARGS.usProtCircuitID`	*n/a*
`DLI_OPT_ARGS.usProtSessionID`	*Protocol Session ID*
`DLI_OPT_ARGS.usProtSequence`	*n/a*
`DLI_OPT_ARGS.usProtXParms [0]`	*n/a*
`DLI_OPT_ARGS.usProtXParms [1]`	*n/a*

The response to the unbind is a read with a call to `dlRead`. If the unbind was successful, the optional argument structure in the response is as follows:

`DLI_OPT_ARGS.usFWPacketType`	*n/a*
`DLI_OPT_ARGS.usFWCommand`	*n/a*
`DLI_OPT_ARGS.usFWStatus`	*n/a*
`DLI_OPT_ARGS.usICPClientID`	*n/a*
`DLI_OPT_ARGS.usICPServerID`	*n/a*
`DLI_OPT_ARGS.usICPCommand`	`DLI_ICP_CMD_UNBIND`
`DLI_OPT_ARGS.ilCPStatus`	`DLI_ICP_ERR_NO_ERR`
`DLI_OPT_ARGS.usICPParms [0]`	*n/a*
`DLI_OPT_ARGS.usICPParms [1]`	*n/a*
`DLI_OPT_ARGS.usICPParms [2]`	*n/a*
`DLI_OPT_ARGS.usProtCommand`	`DLI_ICP_CMD_UNBIND`
`DLI_OPT_ARGS.iProtModifier`	*n/a*
`DLI_OPT_ARGS.usProtLinkID`	*Link stopped*
`DLI_OPT_ARGS.usProtCircuitID`	*n/a*
`DLI_OPT_ARGS.usProtSessionID`	*Protocol Session ID*
`DLI_OPT_ARGS.usProtSequence`	*n/a*
`DLI_OPT_ARGS.usProtXParms [0]`	*n/a*
`DLI_OPT_ARGS.usProtXParms [1]`	*n/a*

The `unbind` command stops an ICP link.

From the ICP's perspective, when the application sends an `unbind` command, the ICP immediately terminates all receiving and transmitting on the link. Deactivation means that data structures are initialized and the link's serial transmitter and receiver are disabled.

For the `unbind` command, the fields of the ICP and protocol headers that the ICP receives contain the following values:

```
ICP_HDR
count       = size of icp header
command     = DLI_ICP_CMD_UNBIND
status      = high bits indicate byte ordering

PROT_HDR
session     = session ID
```

The constant `DLI_ICP_CMD_UNBIND` is in the `command` field of the ICP header and a session number is in the `session` field of the protocol header. The ICP stops devices for that link, clears the link control table link active flag, and returns an acknowledgment to the client. If the link was inactive, the acknowledgment contains an error code in the ICP header's `status` field and the protocol header's `modifier` field. Otherwise they contain zero, indicating success.

8.5.7 Writing to an ICP Link

Once the application has issued a `bind` command to the ICP, it can send messages to the ICP for transmission to the wide-area network. When the ICP receives a message from the client for transmission, it prepares it as required and sends it on the specified link. When the last character is transmitted, the ICP sends a message to the application. The message written by the ICP to the client is called an acknowledgment, however, in this case "acknowledgment" means that the client's message has been transmitted and the memory buffer containing the message has been freed for reuse. It does not mean that the opposite end of the network has acknowledged that it correctly received the message. This is an important area of wide-area communications. It is vital to determine which system is responsible for maintaining a message in case the ultimate end reader does not receive it and the message must be retransmitted. The ICP does not have a disk, and may not be the best platform for maintaining an extensive queue of messages.

8.5.7.1 Writing the Link Configuration to the ICP

To set the link configuration options, a buffer containing the structure defined below is sent to the ICP. The fields of the data structure are set to appropriate values by the client application.

```
/* Structure of configuration request message */

struct conf_type
{
    bit8    protocol;   /* 0 = BSC, 1 = Async, 2 = SDLC     */
    bit8    clock;      /* 0 = external, 1 = internal clock */
    bit8    baud_rate;  /* index into baudsc or baudas      */
    bit8    encoding;   /* 0 = NRZ, 1 = NRZI (SDLC only)    */
    bit8    electrical; /* electrical protocol icp2424      */
    bit8    parity;     /* 0 = none, 1 = odd, 2 = even      */
    bit8    char_len;   /* 7 = 7 bits, 8 = 8 bits           */
                        /*      (asynch only)               */
    bit8    stop_bits;  /* 1 = 1 stop bit, 2 = 2 stop bits  */
                        /*      (asynch only)               */
    bit8    crc;        /* 0 = no CRC, 1 = CRC              */
                        /*      (SDLC always uses CRC)      */
    bit8    syncs;      /* # of leading sync chars (1-8)    */
                        /*      (BSC only)                  */
    bit8    start_char; /* block start character            */
                        /*      (not used for SDLC)         */
    bit8    stop_char;  /* block end char (asynch only)     */
};
typedef struct conf_type CONF_TYPE;
```

Once the client has issued an `attach` command to the ICP, but before it issues a `bind` command, it can send ICP link configuration values to the ICP. If no configuration message is received by the ICP, the default link configuration is used. When the ICP receives a configuration message, it validates it and updates the current link configuration. First the client allocates a buffer for the `CONF_TYPE` structure and fills in the structure. Next the client establishes a `DLI_OPT_ARGS` structure requesting the write, then sends the structure along with a buffer containing the configuration to the ICP. See Section 9.1.1 on page 152 for more information on link configuration.

At the ICP, the fields of the ICP and protocol headers that the ICP receives contain the following values:

```
ICP_HDR
count       = size of icp header and data area
command     = DLI_ICP_CMD_WRITE
status      = high bits indicate byte ordering

PROT_HDR
command     = DLI_PROT_CFG_LINK
session     = session ID
DATA AREA   = configuration data structure
```

The ICP returns these headers to the client as an acknowledgment that the link configuration was completed. The values in the headers of this acknowledgment are the same as those that were received at the ICP, except that the ICP header's `status` field and the protocol header's `modifier` field are filled in with codes reflecting the success of the transaction. The client application receives this acknowledgment by issuing a `read` command as described in Section 8.5.8 on page 145.

8.5.7.2 Writing a Request For Link Statistics From the ICP

The `get statistics` command requests a configuration report for a particular link. The ICP maintains a set of statistics for each link that keeps track of events occurring on each ICP physical port. The client application receives this report by following the `write` command with a `read` command. The `read` command is described in Section 8.5.8 on page 145.

At the ICP, the fields of the ICP and protocol headers that the ICP receives contain the following values:

```
ICP_HDR
count       = size of icp header
command     = DLI_ICP_CMD_WRITE
status      = high bits indicate byte ordering

PROT_HDR
command     = DLI_PROT_GET_STATISTICS_REPORT
session     = session ID
```

The ICP copies the statistics portion of that link's link control table to the data area appended to the protocol header. The ICP then returns this message to the client with the ICP header's `status` field and the protocol header's `modifier` field filled in with a zero to indicate success. The ICP always returns a report, even if that link has not yet been enabled. The client receives the statistics report by issuing a `read` command as described in Section 8.5.8 on page 145. (Note that this statistics report serves as an acknowledgment to the `write` command.) The format of the status report is described in Section 8.5.8.

8.5.7.3 Writing Data to an ICP Link

The `write` command provides data to the ICP for transmission on the specified link. The client establishes a `DLI_OPT_ARGS` structure requesting the write, then sends the structure along with a buffer containing the data to the ICP by calling the DLI `dlWrite` function. At the ICP, the fields of the ICP and protocol headers that the ICP receives contain the following values:

```
ICP_HDR
count       = size of icp header
command     = DLI_ICP_CMD_WRITE
status      = high bits indicate byte ordering

PROT_HDR
command     = DLI_PROT_SEND_NORM_DATA
session     = session ID
DATA AREA   = data to be transmitted
```

The ICP protocol task prepares the data for transmission (for example, calculates the CRC values, and so on), then activates the transmit device for the protocol being used by that link. Once the ICP determines that the data has been sent out over the link, it prepares an acknowledgment that it sends to the client application. The headers of this acknowledgment are of the same form as those described under ICP_HDR and PROT_HDR above, except that the `status` field of the ICP header and the `modifier` field of the protocol header contain zero, reflecting the successful completion of the transmission. The client application receives this acknowledgment by issuing a `read` command as described in Section 8.5.8.

8.5.8 Reading from the ICP Link

The ICP sends three types of messages to the client: command acknowledgments, link statistics, and data read from the wide area network. The client calls the DLI `dlRead` function to access these messages. The application determines the content of the read buffer by examining the `usProtCommand` field of the `DLI_OPT_ARGS` data structure. If the value is `DLI_PROT_SEND_NORM_DATA`, the buffer contains data read from the wide-area network. If the value is `DLI_PROT_GET_STATISTICS_REPORT`, the buffer contains link statistics. If the value is `DLI_PROT_RESP_LOCAL_ACK`, the SPS is writing an acknowledgment to a command.

The application allocates a `DLI_OPT_ARGS` structure, then provides the address of the structure along with the address of a pointer to a buffer to the DLI `dlRead` function.

8.5.8.1 Reading ICP Statistics

The `get statistics` command described in Section 8.5.7.2 on page 143 causes the ICP to send the link statistics to the client application. This report is for the link corresponding to the `usProtLinkID` field in `DLI_OPT_ARGS`. The client application receives the report by issuing a `read` command using the same session ID that was used for the `write` command. The link statistics take the following format:

```
struct STATA
{
   bit16 msg_too_long;
   bit16 dcd_lost;
   bit16 abort_rcvd;
   bit16 rcv_ovrrun;
   bit16 rcv_crcerr;
   bit16 rcv_parerr;
   bit16 rcv_frmerr;
   bit16 xmt_undrun;
   bit16 frame_sent;
   bit16 frame_rcvd;
};
```

At the ICP, the fields of the ICP and protocol headers that the ICP sends to the client application contain the following values:

```
ICP_HDR
count       = size of icp header plus size of data
command     = DLI_ICP_CMD_READ
status      = 0 (success) or an error code

PROT_HDR
command     = DLI_PROT_GET_STATISTICS_REPORT
modifier    = 0 (success) or an error code
session     = session ID
DATA AREA   = statistics report
```

The ICP copies the link control table's statistics data structure to the data area that follows the protocol header and writes error codes to the `status` and `modifier` fields.

8.5.8.2 Reading Normal Data

The read command receives normal data that has arrived on one of the ICP ports. The client issues a read as described in Section 8.5.8 on page 145. The fields of the ICP and protocol headers that the ICP sends to the client application contain the following values:

```
ICP_HDR
count       = size of icp header plus size of data
command     = DLI_ICP_CMD_READ
status      = 0 (success) or an error code

PROT_HDR
command     = DLI_PROT_SEND_NORM_DATA
modifier    = 0 (success) or an error code
session     = session ID
DATA AREA   = incoming data
```

The ICP puts the data read from the link in the area that follows the protocol header.

8.6 Additional Command Types Supported by the SPS

In addition to the API function calls described in the preceding sections, the SPS supports a few commands that are used by layers that lie between the SPS and API layers. These commands are described in the following sections.

8.6.1 Internal Termination Message

The `dl_term` function call is used by a client application when it loses its connection to an ICP. The API issues the `dl_term` function call and provides the return node number to be terminated. The SPS responds by clearing link active flags, turning off devices, and freeing session numbers for all links that had been communicating with the client application using that particular return node number.

The following is an example of the format of the ICP and protocol headers received by the SPS:

```
ICP_HDR
count       = size of icp header
command     = DLI_ICP_CMD_TERM
params[0]   = node number to be terminated (ACK returns on this
              node as well)

PROT_HDR
command     = DLI_ICP_CMD_TERM
```

After performing the `dl_term` call, the SPS returns an acknowledgment on the node that was just terminated. This tells `msgmux` or the ICP driver that the `dl_term` call completed successfully and the node number can be reused.

8.6.2 Internal Test Message

The `dl_test` command is a diagnostic tool used by the client application. Data is written to the SPS in the data area following a protocol header. The utility task immediately returns this data to the client application.

The following is an example of the format of the ICP and protocol headers received by the SPS:

```
ICP_HDR
count       = size of icp header plus size of data
command     = DLI_ICP_CMD_WRITE_EXP
params[0]   = return node number assigned by msgmux or the ICP driver

PROT_HDR
command     = DLI_ICP_CMD_TEST
DATA AREA   = sample data
```

8.6.3 Internal Ping

The `dl_ping` command provides a way for the client application to verify that the SPS is up and running. This message is passed from the utility task to the protocol task before being returned to the client application as an acknowledgment.

The following is an example of the format of the ICP and protocol headers received by the SPS:

```
ICP_HDR
count       = size of icp header
command     = DLI_ICP_CMD_PING
params[0]   = return node number assigned by msgmux or the ICP driver

PROT_HDR
command     = DLI_ICP_CMD_PING
```

Chapter

9 Messages Exchanged between Client and ICP

In messages sent from the client to the ICP, the `usProtCommand` field of the DLI optional argument structure can assume the following values:

`DLI_PROT_CFG_LINK` — send a link configuration message

`DLI_PROT_GET_STATISTICS` — request a report on a link's statistics

`DLI_PROT_SEND_NORM_DATA` — send data to the ICP for transmission

The following sections describe these messages in detail.

9.1 Messages Sent From Client to the ICP

9.1.1 DLI_PROT_CFG_LINK – Client Link Configuration Request

The client sends this message to configure an ICP link. The expected response is a `DLI_PROT_CFG_LINK` message with the `ilCPStatus` field set to `DLI_ICP_ERR_NO_ERR`. If the ICP discovers an error in the message, it returns the configuration message with the `ilCPStatus` field set to `DLI_ICP_ERR_BAD_PARMS`. The optional arguments structure for the configuration message is shown below.

`DLI_OPT_ARGS.usFWPacketType`	`FW_DATA`
`DLI_OPT_ARGS.usFWCommand`	`FW_ICP_WRITE`
`DLI_OPT_ARGS.usFWStatus`	*n/a*
`DLI_OPT_ARGS.uslCPClientID`	*n/a*
`DLI_OPT_ARGS.uslCPServerID`	*n/a*
`DLI_OPT_ARGS.uslCPCommand`	`DLI_ICP_CMD_WRITE`
`DLI_OPT_ARGS.ilCPStatus`	*n/a*
`DLI_OPT_ARGS.uslCPParms [0]`	*n/a*
`DLI_OPT_ARGS.uslCPParms [1]`	*n/a*
`DLI_OPT_ARGS.uslCPParms [2]`	*n/a*
`DLI_OPT_ARGS.usProtCommand`	`DLI_PROT_CFG_LINK`
`DLI_OPT_ARGS.iProtModifier`	*n/a*
`DLI_OPT_ARGS.usProtLinkID`	*Link ICP is to configure.*
`DLI_OPT_ARGS.usProtCircuitID`	*n/a*
`DLI_OPT_ARGS.usProtSessionID`	*Session ID. (See attach message.)*
`DLI_OPT_ARGS.usProtSequence`	*n/a*
`DLI_OPT_ARGS.usProtXParms [0]`	*n/a*
`DLI_OPT_ARGS.usProtXParms [1]`	*n/a*

The data area in the write is an instance of the CONF_TYPE structure. This structure is defined as follows:

```
typedef struct {
unsigned char protocol;      /* 0 = bsc, 1 = async, 2 = sdlc */
unsigned char clock;         /* (bsc and sdlc only)
                                0 = external, 1 = internal   */
unsigned char baud_rate;     /* For protocol = async, the following
                                values apply:
                                  0 =    300
                                  1 =    600
                                  2 =   1200
                                  3 =   1800
                                  4 =   2400
                                  5 =   3600
                                  6 =   4800
                                  7 =   7200
                                  8 =   9600
                                  9 =  19200
                                 10 =  38400
                                 11 =  57600
                                 12 = 115000
                                 13 = 230400
                                For bsc and sdlc, the following
                                values apply:
                                  0 =     300
                                  1 =     600
                                  2 =    1200
                                  3 =    2400
                                  4 =    4800
                                  5 =    9600
                                  6 =   19200
                                  7 =   38400
                                  8 =   57600
                                  9 =   64000
                                 10 =  307000
                                 11 =  460800
                                 12 =  614400
                                 13 =  737300
                                 14 =  921600
                                 15 = 1228800
                                 16 = 1843200
                                                              */
```

```
unsigned char encoding;     /* (sdlc only) 0 = NRZ, 1 = NRZI   */
unsigned char unused;       /* not used                        */
unsigned char parity;       /* (async only) 0 = none, 1 = odd,
                               2 = even                        */
unsigned char char_len;     /* (async only) 7 = 7 bits,
                               8 = 8 bits                      */
unsigned char stop_bits;    /* (async only) 1 = 1 stop bit,
                               2 = 2 stop bits                 */
unsigned char crc;          /* (async and bsc only)
                               0 = no CRC, 1 = CRC             */
unsigned char syncs;        /* (bsc only) number of leading
                               sync characters 1 to 8          */
unsigned char start_char;   /* (async and bsc only) block
                               start character                 */
unsigned char stop_char;    /* (async only) block end
                               character                       */
};
```

9.1.2 DLI_PROT_GET_STATISTICS – Client Link Statistics Request

The client sends this message to request the statistics on an ICP link. The expected response is a DLI_PROT_GET_STATISTICS_REPORT message with the iICPStatus field set to DLI_ICP_ERR_NO_ERR. If the ICP discovers an error in the message, it returns the statistics request message with the iICPStatus field set to an error code. The DLI_OPT_ARGS structure for the configuration message is shown below.

DLI_OPT_ARGS.usFWPacketType	FW_DATA
DLI_OPT_ARGS.usFWCommand	FW_ICP_WRITE
DLI_OPT_ARGS.usFWStatus	*n/a*
DLI_OPT_ARGS.usICPClientID	*n/a*
DLI_OPT_ARGS.usICPServerID	*n/a*
DLI_OPT_ARGS.usICPCommand	DLI_ICP_CMD_WRITE
DLI_OPT_ARGS.iICPStatus	*n/a*
DLI_OPT_ARGS.usICPParms [0]	*n/a*
DLI_OPT_ARGS.usICPParms [1]	*n/a*
DLI_OPT_ARGS.usICPParms [2]	*n/a*
DLI_OPT_ARGS.usProtCommand	DLI_PROT_GET_STATISTICS_REPORT
DLI_OPT_ARGS.iProtModifier	*n/a*
DLI_OPT_ARGS.usProtLinkID	*Link ICP is to report on.*
DLI_OPT_ARGS.usProtCircuitID	*n/a*
DLI_OPT_ARGS.usProtSessionID	*Session ID. (See attach message.)*
DLI_OPT_ARGS.usProtSequence	*n/a*
DLI_OPT_ARGS.usProtXParms [0]	*n/a*
DLI_OPT_ARGS.usProtXParms [1]	*n/a*

There is no data area for a link statistics request.

9.1.3 DLI_PROT_SEND_NORM_DATA – Client Send ICP Link Data

The client sends this message to request the ICP to transmit data on a link. The expected response is a `DLI_PROT_RESP_LOCAL_ACK` message when the message has been transmitted by the ICP.

`DLI_OPT_ARGS.usFWPacketType`	`FW_DATA`
`DLI_OPT_ARGS.usFWCommand`	`FW_ICP_WRITE`
`DLI_OPT_ARGS.usFWStatus`	*n/a*
`DLI_OPT_ARGS.usICPClientID`	*n/a*
`DLI_OPT_ARGS.usICPServerID`	*n/a*
`DLI_OPT_ARGS.usICPCommand`	`DLI_ICP_CMD_WRITE`
`DLI_OPT_ARGS.ilCPStatus`	*n/a*
`DLI_OPT_ARGS.usICPParms [0]`	*n/a*
`DLI_OPT_ARGS.usICPParms [1]`	*n/a*
`DLI_OPT_ARGS.usICPParms [2]`	*n/a*
`DLI_OPT_ARGS.usProtCommand`	`DLI_PROT_SEND_NORM_DATA`
`DLI_OPT_ARGS.iProtModifier`	*n/a*
`DLI_OPT_ARGS.usProtLinkID`	*Link ICP is to transmit on.*
`DLI_OPT_ARGS.usProtCircuitID`	*n/a*
`DLI_OPT_ARGS.usProtSessionID`	*Session ID. (See attach message.)*
`DLI_OPT_ARGS.usProtSequence`	*n/a*
`DLI_OPT_ARGS.usProtXParms [0]`	*n/a*
`DLI_OPT_ARGS.usProtXParms [1]`	*n/a*

The data area in the write is the data to be transmitted. If the protocol is bsc, the configured number of sync characters are placed at the beginning of the message. If the protocol is async or bsc, the configured start character is placed immediately before the client's data. If the protocol is async, the configured stop character is appended. If a CRC is configured, a CRC is calculated and appended.

9.2 Messages Sent From ICP To Client

9.2.1 DLI_PROT_CFG_LINK – ICP Acknowledge Link Configuration

The ICP sends this message to acknowledge that the client's link configuration request was performed.

DLI_OPT_ARGS.usFWPacketType	*n/a*
DLI_OPT_ARGS.usFWCommand	*n/a*
DLI_OPT_ARGS.usFWStatus	*n/a*
DLI_OPT_ARGS.usICPClientID	*n/a*
DLI_OPT_ARGS.usICPServerID	*n/a*
DLI_OPT_ARGS.usICPCommand	*n/a*
DLI_OPT_ARGS.iICPStatus	DLI_ICP_ERR_NO_ERR
DLI_OPT_ARGS.usICPParms [0]	*n/a*
DLI_OPT_ARGS.usICPParms [1]	*n/a*
DLI_OPT_ARGS.usICPParms [2]	*n/a*
DLI_OPT_ARGS.usProtCommand	DLI_PROT_CFG_LINK
DLI_OPT_ARGS.iProtModifier	*n/a*
DLI_OPT_ARGS.usProtLinkID	*Link ICP configured.*
DLI_OPT_ARGS.usProtCircuitID	*n/a*
DLI_OPT_ARGS.usProtSessionID	*Session ID. (See attach message.)*
DLI_OPT_ARGS.usProtSequence	*n/a*
DLI_OPT_ARGS.usProtXParms [0]	*n/a*
DLI_OPT_ARGS.usProtXParms [1]	*n/a*

The data area is not applicable.

9.2.2 DLI_PROT_GET_STATISTICS – ICP Statistics Report

The ICP sends this message to report the statistics on an ICP link. The DLI_OPT_ARGS structure for the configuration message is shown below.

DLI_OPT_ARGS.usFWPacketType	*n/a*
DLI_OPT_ARGS.usFWCommand	*n/a*
DLI_OPT_ARGS.usFWStatus	*n/a*
DLI_OPT_ARGS.usICPClientID	*n/a*
DLI_OPT_ARGS.usICPServerID	*n/a*
DLI_OPT_ARGS.usICPCommand	*n/a*
DLI_OPT_ARGS.ilCPStatus	DLI_ICP_ERR_NO_ERR
DLI_OPT_ARGS.usICPParms [0]	*n/a*
DLI_OPT_ARGS.usICPParms [1]	*n/a*
DLI_OPT_ARGS.usICPParms [2]	*n/a*
DLI_OPT_ARGS.usProtCommand	DLI_PROT_GET_STATISTICS_REPORT
DLI_OPT_ARGS.iProtModifier	*n/a*
DLI_OPT_ARGS.usProtLinkID	*Link ICP is reporting on.*
DLI_OPT_ARGS.usProtCircuitID	*n/a*
DLI_OPT_ARGS.usProtSessionID	*Session ID. (See attach message.)*)
DLI_OPT_ARGS.usProtSequence	*n/a*
DLI_OPT_ARGS.usProtXParms [0]	*n/a*
DLI_OPT_ARGS.usProtXParms [1]	*n/a*

The data area for the link statistics report contains the structure SPS_STATS_REPORT. This structure is defined as follows:

```
typedef struct {
     bit16   msg_too_long;  Number of messages read from WAN
                            and thrown away
     bit16   dcd_lost;      Number of times receiver restarted
                               because carrier was lost
     bit16   abort_rcvd;    Number of times receiver restarted
                               because abort was received
     bit16   rcv_ovrrun;    Number of messages received with receiver overruns
     bit16   rcv_crcerr;    Number of messages received with bad CRCs
     bit16   rcv_parerr;    Async only. Parity errors
     bit16   rcv_frmerr;    Async only. Number of framing errors
     bit16   xmt_undrun;    Number of transmit underruns
     bit16   frame_sent;    Number of message buffers sent
     bit16   frame_rcvd;    Number of message buffers received
} SPS_STATS_REPORT;
```

9.2.3 DLI_PROT_SEND_NORMAL_DATA – ICP Send Data To Client

The ICP sends this message to the client to provide it with messages read from the link. The DLI_OPT_ARGS structure for the DLI_ICP_CMD_READ message is shown below.

DLI_OPT_ARGS.usFWPacketType	*n/a*
DLI_OPT_ARGS.usFWCommand	*n/a*
DLI_OPT_ARGS.usFWStatus	*n/a*
DLI_OPT_ARGS.usICPClientID	*n/a*
DLI_OPT_ARGS.usICPServerID	*n/a*
DLI_OPT_ARGS.usICPCommand	DLI_ICP_CMD_READ
DLI_OPT_ARGS.iICPStatus	*n/a*
DLI_OPT_ARGS.usICPParms [0]	*n/a*
DLI_OPT_ARGS.usICPParms [1]	*n/a*
DLI_OPT_ARGS.usICPParms [2]	*n/a*
DLI_OPT_ARGS.usProtCommand	DLI_PROT_SEND_NORM_DATA
DLI_OPT_ARGS.iProtModifier	*n/a*
DLI_OPT_ARGS.usProtLinkID	*Link ICP read data from.*
DLI_OPT_ARGS.usProtCircuitID	*n/a*
DLI_OPT_ARGS.usProtSessionID	*Session ID. (See attach message.)*
DLI_OPT_ARGS.usProtSequence	*n/a*
DLI_OPT_ARGS.usProtXParms [0]	*n/a*
DLI_OPT_ARGS.usProtXParms [1]	*n/a*

The data area contains the data read. The stop character and CRC characters are not provided.

9.2.4 DLI_PROT_RESP_LOCAL_ACK – ICP Acknowledge Message

The ICP sends this message to inform the client that the ICP has completed transmission of client's message. The `DLI_OPT_ARGS` structure for the `DLI_PROT_RESP_LOCAL_ACK` message is shown below.

`DLI_OPT_ARGS.usFWPacketType`	*n/a*
`DLI_OPT_ARGS.usFWCommand`	*n/a*
`DLI_OPT_ARGS.usFWStatus`	*n/a*
`DLI_OPT_ARGS.uslCPClientID`	*n/a*
`DLI_OPT_ARGS.uslCPServerID`	*n/a*
`DLI_OPT_ARGS.uslCPCommand`	*n/a*
`DLI_OPT_ARGS.ilCPStatus`	*n/a*
`DLI_OPT_ARGS.uslCPParms [0]`	*n/a*
`DLI_OPT_ARGS.uslCPParms [1]`	*n/a*
`DLI_OPT_ARGS.uslCPParms [2]`	*n/a*
`DLI_OPT_ARGS.usProtCommand`	`DLI_PROT_RESP_LOCAL_ACK`
`DLI_OPT_ARGS.iProtModifier`	*n/a*
`DLI_OPT_ARGS.usProtLinkID`	*Link ICP transmitted data on.*
`DLI_OPT_ARGS.usProtCircuitID`	*n/a*
`DLI_OPT_ARGS.usProtSessionID`	*Session ID. (See attach message.)*
`DLI_OPT_ARGS.usProtSequence`	*n/a*
`DLI_OPT_ARGS.usProtXParms [0]`	*n/a*
`DLI_OPT_ARGS.usProtXParms [1]`	*n/a*

There is no data area.

Appendix

A Application Notes

This appendix clarifies some points made in the technical manuals and describes some peculiarities of the devices and the ICP hardware.

- When programming in a high-level language, be sure that your compiler's optimizer handles the special requirements of device-level programming correctly. For example, if you program two writes to a hardware register in sequence, the optimizer could inappropriately remove the first write instruction as superfluous.

Appendix

B Data Rate Time Constants for IUSC Programming

This appendix provides some commonly used baud rate time constants for IUSC programming on the ICP.

Table B–1 shows IUSC time constants for 1X mode for the ICP2432B normally used for all synchronous communication modes. Table B–2 shows IUSC time constants for 16X mode for the ICP2432B normally used for asynchronous mode. Set the required clock mode in the Channel Mode register of the IUSC.

The IUSC time constant is a 16-bit value stored in Time Constant register 0 or 1.

Table B–1: IUSC Time Constants for 1X Clock Rate for ICP2432B

Baud Rate (kbits/sec)	Time Constant (hexadecimal)
0.3	2FFF
0.6	17FF
1.2	0BFF
2.4	05FF
4.8	02FF
9.6	017F
19.2	00BF
38.4	005F
57.6	003F

Table B–2: IUSC Time Constants for 16X Clock Rate for ICP2432B

Baud Rate (kbits/sec)	**Time Constant (hexadecimal)**
0.3	02FF
0.6	017F
1.2	00BF
2.4	005F
4.8	002F
9.6	0017
19.2	000B
38.4	0005
57.6	0003

Appendix

C Error Codes

There are several methods used by the DLI and ICP software to report errors, as described in the following sections:

C.1 DLI Error Codes

The error code can be returned directly by the DLI function call in the global variable `dlerrno`. Typical errors are those described in the *Freeway Data Link Interface Reference Guide*.

C.2 ICP Global Error Codes

Table C–1 lists the ICP-related errors that can be returned in the global variable `iICPStatus`. The `DLI` constants are defined in the `dlicperr.h` file.

C.3 ICP Error Status Codes

The ICP-related errors listed in Table C–1 can also be returned in the `dlRead` `optArgs.iICPStatus` field of the response, which is a duplicate of the `iIPCStatus` global variable. The DLI sets the `dlRead` `optArgs.usProtCommand` field to the same value as the `dlWrite` request that caused the error.

Table C–1: ICP Error Status Codes used by the ICP

Code	Mnemonic	Meaning
0	DLI_ICP_ERR_NO_ERR	A data block has been successfully transmitted or received on the line or a command has been successfully executed.
–101	DLI_ICP_ERR_BAD_NODE	An invalid node number was passed to the ICP from the DLI.
–102	DLI_ICP_ERR_BAD_LINK	The link number from the client program is not a legal value.
–103	DLI_ICP_ERR_NO_CLIENT	The maximum number of clients are registered for the link.
–105	DLI_ICP_ERR_BAD_CMD	The command from the client program is not a legal value.
–115	DLI_ICP_ERR_BUF_TOO_SMALL	The size of the data buffer sent from the client exceeds the size of the configured buffers.
–117	DLI_ICP_ERR_LINK_ACTIVE	A client request to enable (bind) a link is rejected by the ICP because the link is already enabled.
–118	DLI_ICP_ERR_LINK_INACTIVE	A client request to disable (unbind) a link is rejected by the ICP because the link is already disabled.
–119	DLI_ICP_ERR_BAD_SESSID	The session identification is invalid.
–121	DLI_ICP_ERR_NO_SESSION	A client request to attach a link is rejected by the ICP because the session identification is invalid.
–122	DLI_ICP_ERR_BAD_PARMS	The values used for the function call are illegal.
–145	DLI_ICP_ERR_INBUF_OVERFLOW	Server buffer input overflow
–146	DLI_ICP_ERR_OUTBUF_OVERFLOW	Server buffer output overflow

Appendix

D Test Programs

The Software Protocol Toolkit loopback test programs[1] and test directories are listed in Table D–1 for UNIX systems, Table D–2 for VMS systems, and Table D–3 for Windows NT systems. This section gives a summary of the steps required to run the loopback test; see the *Freeway Server User's Guide* or the appropriate Freeway embedded user's guide for details and an example output. The I/O (blocking or non-blocking) is selected using the `asyncIO` DLI configuration parameter (described in the *Freeway Data Link Interface Reference Guide*) which defaults to "no" (blocking I/O).

Table D–1: UNIX Loopback Test Programs and Directories

Loopback Program	Type of I/O	UNIX Test Directory
`spsslp.c`	Blocking I/O	`usr/local/freeway/client/test/sps`
`spsalp.c`	Non-blocking I/O	`usr/local/freeway/client/test/sps`

Table D–2: VMS Loopback Test Programs and Directories

Loopback Program	Type of I/O	VMS Test Directory
`SPSSLP.C`	Blocking I/O	`SYS$SYSDEVICE:[FREEWAY.CLIENT.TEST.SPS]`
`SPSALP.C`	Non-blocking I/O	`SYS$SYSDEVICE:[FREEWAY.CLIENT.TEST.SPS]`

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

Table D–3: Windows NT Loopback Test Program and Directory

Loopback Program	Type of I/O	Windows NT Test Directory
`spsslp.c`	Blocking I/O	`c:\freeway\client\test\sps`
`spsalp.c`	Non-blocking I/O	`c:\freeway\client\test\sps`

To run one of the test programs, perform the following steps:

1. Make sure the `server` TSI configuration parameter is correctly defined in the TSI text configuration file for each TSI connection definition. Refer to the *Freeway Transport Subsystem Interface Reference Guide*.

2. Make any required changes to the DLI text configuration file for DLI session parameters or ICP link parameters whose values differ from the defaults. Refer to the *Freeway Data Link Interface Reference Guide*.

3. Be sure you are in the correct directory.

   For UNIX: `cd /usr/local/freeway/client/test/sps`
   For VMS: `SET DEF SYS$SYSDEVICE:[FREEWAY.CLIENT.TEST.SPS]`
   For NT: `cd c:\freeway\client\test\sps`

4. Run the make file provided in the test directory.

   For UNIX: `make -f makefile.`<*op-sys*>` all`
   where <*op-sys*> is the operating system:
   `dec` (for a DEC UNIX system)
   `hpux` (for an HP/UX system)
   `sol` (for a Solaris system)
   `sun` (for a Sun system)
   For example: `make -f makefile.sun all`

   For VMS: `@MAKEVMS "" UCX`

For Windows NT (Freeway server): `nmake -f makefile.<op-sys> all`

where <*op-sys*> is the operating system:

`ant` (for Alpha NT)

`int` (for Intel NT)

For example: `nmake -f makefile.ant all`

For Windows NT (embedded): `nmake -f makefile.nti`

The make file automatically performs the following:

- In VMS systems only, creates the foreign commands used for the `dlicfg` and `tsicfg` configuration preprocessor programs. (This is not necessary for UNIX and NT systems.)

- In all systems, runs the `dlicfg` and `tsicfg` configuration preprocessor programs. These programs process the appropriate DLI and TSI text configuration files to create the DLI and TSI binary configuration files. The text configuration files provided for blocking and non-blocking I/O are:

	Blocking I/O	**Non-blocking I/O**
DLI:	`spssldcfg`	`spsaldcfg`
TSI:	`spssltcfg`	`spsaltcfg`

  The resulting binary configuration files have the same names with a `.bin` extension. For example, `spssldcfg.bin`.

- In all systems, copies the DLI and TSI binary configuration files to the appropriate `bin` directory.

  UNIX example: `freeway/client/`*op-sys*`/bin`

  VMS example: `[FREEWAY.CLIENT.`<*vms_platform*>`_UCX.BIN]`
  where <*vms_platform*> is `VAX` or `AXP`
  for example, `[FREEWAY.CLIENT.VAX_UCX.BIN]`

NT example: `freeway\client\`*op-sys*`\bin`

where <*op-sys*> is the operating system:

`ant`	(for Alpha NT)
`int`	(for Intel NT)
`int_nt_emb`	(for embedded)

- In all systems, compiles and links the loopback test programs (`spsslp.c` and/or `spsalp.c`) and copies them to the same `bin` directory.

5. Boot the Freeway server or run `icpload` on the embedded product to download the SPS software onto the ICP.

6. Connect two ICP links with loopback cables.

7. Execute the test program from the directory where the *binary* DLI and TSI configuration files reside (that resulted from Step 4 above).

In Step 4 above, the make file runs the `dlicfg` and `tsicfg` preprocessor programs *and* compiles and links the test programs. If you already compiled and linked the test programs, you can avoid recompiling and relinking them by running `dlicfg` and `tsicfg` yourself instead of running the make file. However, note the following if you do.

In a UNIX system, if you run `dlicfg` and `tsicfg` instead of running the make file, you must manually move the resulting DLI and TSI binary configuration files to the appropriate `freeway/client/`*op-sys*`/bin` directory where *op-sys* indicates the operating system: `sunos`, `hpux`, `solaris`, `rs_aix`, `osf1`. For example, `freeway/client/sunos/bin`.

In a VMS system, if you run `dlicfg` and `tsicfg` instead of running the make file, you must do the following:

- *Before* you run `dlicfg` and `tsicfg`, run the `makefc.com` command file to create the foreign commands used for `dlicfg` and `tsicfg`.

```
@MAKEFC UCX
```

- *After* you run `dlicfg` and `tsicfg`, run the `move.com` command file which moves the DLI and TSI binary configuration files to the `bin` directory for your TCP/IP package.

  `@MOVE` *filename* `UCX`

  where: *filename* is the name of the binary configuration file

  For example: `@MOVE SPSSLDCFG.BIN UCX`

In a Windows NT system, if you run `dlicfg` and `tsicfg` instead of running the make file, you must manually move the resulting DLI and TSI binary configuration files to the appropriate `freeway\client\`*op-sys*`\bin` directory where *op-sys* indicates the operating system: `ant` or `int` for the Freeway server or `int_nt_emb` for the Freeway embedded product. For example, `freeway\client\ant\bin`.

Index

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