Intelligent I/O (I_2O) Architecture Specification

Draft Revision 1.5

March 1997

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Version 1.5 is being released with some missing details, so that vendors can start implementing sooner and participate in defining future revisions of the I_2O Architecture Specification. Missing Information is noted with TBD (to be defined) and some indication of the current thinking is provided where practical.

The I_2O Special Interest Group ("I_2O SIG") retains the right to make changes to this specification at any time, without notice.

The I_2O SIG disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information in this specification. The I_2O SIG does not warrant or represent that such implementations(s) will not infringe such rights.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

* Third party marks and brands are the property of their respective owners.

Table of Contents

Tables

Figures

1 Introduction

1.1 I$_2$O System Conceptual Overview

The *Intelligent I/O (I$_2$O) Architecture Specification* describes an open architecture for developing device drivers in network system environments. The architecture is independent of the operating system (OS), processor platform, and system I/O bus. This specification strives to standardize development so that portions of the driver can be offloaded to an embedded processor on another hardware platform.

The current trend for high-end networking and storage technology pushes more functionality down to the low-level driver, while demanding higher performance. To meet these requirements, hardware vendors are turning to intelligent products that contain their own I/O controller for processing I/O transactions, such as RAID controllers for storage and ATM controllers for networking.

1.1.1 Benefits

Added intelligence at the hardware level provides many benefits. Using special processors to complete I/O transactions reduces host CPU utilization. It also exports interrupts, which disrupt application processing, to an environment that more efficiently handles I/O's real-time requirements. The goal of the I$_2$O specification goes beyond just adding intelligence, however. It is an effort to standardize on intelligent platforms for the benefit of all segments of the industry and provides the following advantages:

1.1.1.1 Reduced Driver Interfaces

Today, each hardware vendor must supply multiple drivers for each piece of hardware. The number multiplies with the different operating environments and markets. Both the OS and hardware vendors must test and certify many versions of drivers.

The I$_2$O specification enables the OS vendor to produce a single driver for each class of device and concentrate on optimizing the OS portion of the driver. Furthermore, the hardware vendor needs to produce only one version of that driver, which works for any OS that supports I$_2$O. This enables the hardware vendor to focus on its own technology, optimizing that single driver. It allows early market penetration where proliferating drivers once restricted time to market or even prevented entering it at all.

1.1.1.2 Economy of Scale

Besides the system interface for an intelligent I/O device, the I$_2$O specification also defines an operating environment for the I/O subsystem. This enables the system vendor to create an I/O platform that can support a number of non-intelligent I/O adapters, providing economy of scale. Where a single I/O adapter might not justify the additional cost of an I/O processor, combining a number of these adapters under a single I/O platform can produce a cost-effective, intelligent solution.

1.1.1.3 Stacked Drivers

Stacked drivers enable a third-party software vendor to provide value added expansion, independent of both the OS and hardware. The I_2O specification pioneers *peer-to-peer* capability. Peer-to-peer allows data movement between I/O subsystems without affecting the host or taxing its CPU.

To summarize, the benefits of the intelligent I/O architecture include:

- Reduced expense for intelligent I/O adapter card vendors developing and maintaining products that support multiple operating systems.
- Reduced time for intelligent I/O adapter card vendors to bring their family of products to market.
- Accelerated development of technology for I/O solutions, resulting in more innovative solutions for the end user.
- Increased competition, reducing prices for the end user.
- Decoupled I/O adapter hardware and software development.
- Simplified testing during device driver development, because the split driver model isolates functionality.
- Freedom to develop new technology instead of maintaining existing device driver software across systems.
- Enhanced system availability and management, and improved fault isolation and recovery, due to the physical and logical isolation of the I/O subsystem.
- Improved system performance because I/O-intensive functions of the OS are distributed to an intelligent I/O subsystem.
- Extensible execution environment, accommodating new I/O technologies and interconnections.

1.1.2 Features

The benefits of the I_2O specification stem from two basic features:

1. its model for splitting device drivers
2. its standard interfaces between the device drivers, I/O platforms, and host OS.

1.1.2.1 Split Device Driver

A device driver interfaces a particular hardware device to a specific operating system. As shown in Figure 1-1, the top portion of a device driver adapts the operating system calls into I/O transactions. The bottom portion of the driver contains vendor-specific code, which adapts the hardware level interface of the I/O adapter to the functions required for that particular class of device. For example, all network adapters perform the same class-specific functions but have varying register interfaces.

The I_2O specification splits the device driver into two modules: one that contains all the OS-specific code and the other for hardware-specific code. OS vendors need to produce only one OS-specific module for each class of I/O device. Likewise, hardware vendors have to produce only a single version of the hardware device module for an I/O adapter.

The device driver can be split more than once, creating *stackable drivers*. This enables an independent software vendor to support system expansion, independent of both the hardware and the OS.

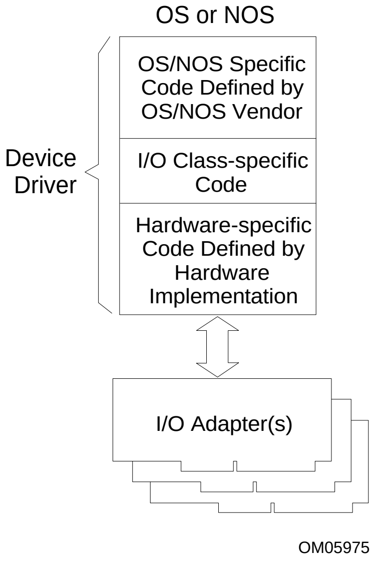

OS or NOS
Device Driver
OS/NOS Specific Code Defined by OS/NOS Vendor
I/O Class-specific Code
Hardware-specific Code Defined by Hardware Implementation
I/O Adapter(s)
OM05975

Figure 1-1. Typical I/O Device Driver

1.1.2.2 I_2O Interfaces

The typical I_2O system consists of a host node and multiple I/O platforms with a variety of I/O adapters. The specification provides standard interfaces for each piece of the system, as illustrated in Figure 1-2.

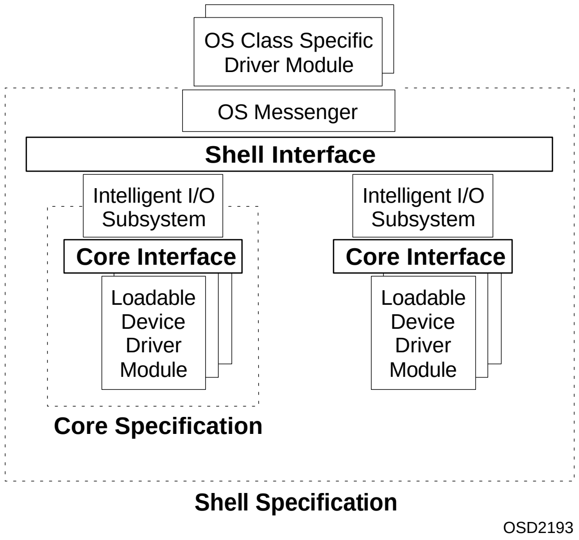

OS Class Specific Driver Module
OS Messenger
Shell Interface
Intelligent I/O Subsystem
Core Interface
Loadable Device Driver Module
Intelligent I/O Subsystem
Core Interface
Loadable Device Driver Module
Core Specification
Shell Specification
OSD2193

Figure 1-2. I_2O Interfaces

The *shell specification* defines the interface that an I/O subsystem presents to the host. It specifies the behavior of both the system and subsystem when initializing and managing intelligent I/O subsystems. The shell interface provides both OS and I/O subsystem independence.

The *core specification* defines the interface between a loadable device driver and the I/O platform. The interface provides an operating environment for device drivers that, like the shell interface, is independent of both the OS and the I/O platform. This enables any real-time operating system to host device drivers produced by third-party hardware vendors.

The *message-based interfaces* enable direct message passing between any two device driver modules for a particular class of I/O (message class). I_2O specifies message classes for each of the following:

- LAN ports, such as Ethernet or Token Ring controllers
- random block storage devices, such as hard disk drives and CD-ROM drives
- sequential storage devices and variable block-size devices, such as tape drives
- SCSI ports
- SCSI devices

Candidates for future message classes include the following:

- IDE controllers and devices
- Floppy disk controllers and devices
- Fibre Channel
- WAN ports, such as ATM controllers

1.2 Objectives

The objectives of this specification include the following:

- To specify an architecture that is operating-system-vendor-independent and adapts to existing operating systems.
- To define an environment that coexists with existing device drivers; legacy device drivers can be ported to the new environment at the vendor's discretion.
- To provide an architecture that isolates the intelligent I/O subsystem from the host operating system. The execution environment created by the architecture enhances system performance and functionality.
- To create an architecture that allows device drivers to scale across system platforms, from high-end workstations to high-end servers.
- To enable device drivers to port across target processors; portability refers to the device driver source code written in ANSI C.

1.3 Scope

Numerous vendors develop intelligent I/O adapters for a variety of platforms and system interconnections. The I_2O Special Interest Group (SIG) supports and encourages vendors to develop advanced I/O approaches that meet the technological requirements of today's computing environments. The *Intelligent I/O Architecture Specification* neither prevents nor discourages industry vendors from adding their own value to I/O systems. It focuses on a standard interface, allowing I/O adapter vendors to develop device drivers more cost-effectively. System integration and OS vendors will recognize opportunities to add value to products within the environment the specification defines.

This document specifies an Intelligent I/O platform (shell specification) composed of the following:

- a register-level interface
- host programming model
- I/O platform behavioral model
- messaging model
- messages specific to each class of I/O service.

This document also specifies an I/O platform internal interface (core specification) between driver modules and the I/O platform. It enables a system vendor to provide a generic platform that can host third-party driver modules. The core specification includes an event-driven model and a set of APIs that provide:

- the operating environment
- a message interface
- a system abstraction
- data transport services.

1.4 Document Organization

Table 1-1 describes the other chapters in this specification.

The I_2O SIG provides, for your convenience, supplemental information to this specification on our Web site. Hardware-specific information is related to, but not part of, this document. Where it is relevant, we will refer you to our Web site at http://www.i2osig.org/

Table 1-1. Chapters in This Document

Chapter	Description
2	**Overview of the I_2O Architecture**: Chapter 2 provides a technical overview of the Intelligent I/O (I_2O) Architecture by describing its concepts and structures. The overview explains the split driver model, which takes advantage of the intelligence provided by a processor in an intelligent I/O subsystem. This chapter also describes the inter-operation of the components and interfaces of the architecture.
3	**Basic Requirements**: Chapter 3 defines the basic requirements for conformance and common structures used throughout the specification.
4	**Shell Specification**: Chapter 4 defines the system interface for the intelligent I/O subsystem, which provides a communication path between the operating system and the I/O driver modules on the I/O platform. The shell specification describes the interface between the host processor and I/O platform. It also specifies the platform-to-platform interface for peer-to-peer operation between I/O platforms.
5	**Core Specification**: Chapter 5 specifies the operating environment within the I/O subsystem for device driver modules and the operational interfaces of those modules.
6	**Class Definitions**: Chapter 6 describes the message definitions for each message class. It also defines all class-dependent structures.
Appendix A	**Differences from Previous Version**: Appendix A describes the differences from version 1.0 of this specification.

1.5 I_2O Include Files

Table 1-2 describes the include files for this specification and their relationship to the appropriate sections of this document. Include files, available in electronic format, help the driver developer incorporate changes and upgrades.

Table 1-2. I$_2$O Shell Include Files

File Name	Chapter-Section	Description
`i2oTypes.h`	n/a	I2O data types
`i2omsg.h`	3.4	Message header definitions
`i2oexec.h`	4.4	IOP executive definitions
`i2ocore.h`	5.x	DDM structure definitions
`i2oirtos.h`	5.4	IRTOS API function definitions
`i2outil.h`	6.1	Utility message definitions
`i2omstor.h`	6.4	Random block storage class message definitions
`i2otstor.h`	6.5	Sequential storage class message definitions
`i2obscsi.h`	6.6	SCSI adapter class and SCSI peripheral class message definitions
`i2oadptr.h`	6.7	Adapter class message definitions
`i2olan.h`	6.10	LAN class message definitions

Table 1-3. I$_2$O Core Include Files (for developing a DDM that runs under an IRTOS)

File Name	Chapter-Section	Description
`i2o.h`	n/a	Generic IRTOS definitions
`i2oPciLib.h`	n/a	PCI definitions
`i2oModule.h`	5.3.2	Module Parameter Block and Header definitions.
`i2oCfgLib.h`	5.3.3	Configuration Dialog definitions
`i2oBusLib.h`	5.4.10	Bus definitions
`i2oAdapterLib.h`	5.4.11	Adapter definitions
`i2oMemSetLib.h`	5.4.12	Memory Set definitions
`i2oPageLib.h`	5.4.12.3	Page Set definitions
`i2oIntLib.h`	5.4.13	Interrupt definitions
`i2oTimerLib.h`	5.4.13	Time definitions
`i2oDmaLib.h`	5.4.15	DMA definitions
`i2oThreadLib.h`	5.4.16-5.4.17	Thread definitions
`i2oSemLib.h`	5.4.18	Semaphore definitions
`i2oPipeLib.h`	5.4.19	Pipe definitions
`i2oErrorLib.h`	5.4.2.1	Error Handling definitions
`i2oObjLib.h`	5.4.2.3	Object definitions
`i2oEvtQLib.h`	5.4.2.4	Event definitions
`i2oIopLib.h`	5.4.20	IOP Information definitions
`i2oDdmLib.h`	5.4.3-5.4.6	DDM and Device definitions
`i2oFrameLib.h`	5.4.8	Frame definitions

1.6 Audience

This document is intended for industry members, and the most relevant information for each can vary, as shown in Table 1-4. All readers should begin with Chapter 2, the system overview, and Chapter 3, which includes common material for all patrons of the I_2O specification. One must read the entire specification to thoroughly understand the architecture.

Table 1-4. Reader's Guide

If you are:	You'll be most interested in:
An independent hardware vendor	Chapter 2, *Technical Overview* Chapter 3, *Basic Requirements* Chapter 6, *Class Specifications*
- for a non-intelligent card plugged into an I/O platform	also read Chapter 5, *I_2O Core Specification*
- for a standalone card with its own processor and integrated controllers	also read Chapter 4, *I_2O Shell Interface Specification*, and, if the card supports stackable drivers, read Chapter 5, *I_2O Core Specification*
A system vendor of an open intelligent I/O platform	Chapter 2, *Technical Overview* Chapter 3, *Basic Requirements* Chapter 4, *I_2O Shell Interface Specification* Chapter 5, *I_2O Core Specification*
An operating system vendor	Chapter 2, *Technical Overview* Chapter 3, *Basic Requirements* Chapter 4, *Executive Messages and Structures* Chapter 6, *Utility Messages*
An independent software vendor of stackable device driver modules	Chapter 2, *Technical Overview* Chapter 3, *Common Facilities and Structures* Chapter 5, *I_2O Core Specification* Chapter 6, *Utility Messages* and messages for specific I/O classes

This specification assumes that you understand the environment of operating systems and the I/O architecture of network systems.

1.7 Definition of Terms

This specification uses the following terms:

Table 1-5. Definitions

Term	Description
Adapter	A set of hardware whose operation accesses one or more I/O devices or ports. An adapter is controlled by either exactly zero or exactly one hardware driver module. An adapter can contain more than one port or function of the same or different class. An adapter is identified by its physical location.
Assigned Adapter	An adapter assigned to an IOP. The IOP controls it. The host accesses the adapter and its devices through the I_2O message service.

Table 1-5. Definitions (continued)

Term	Description
Application Processor	A processing element of a host designed to process diverse application programs.
Batch	The collection of I/O transactions in one operation, such as the LanPacketSend command of the LAN class. A batch is a concept, not a data structure. When the IOP reports status on a batch basis, the status applies to all transactions in the aggregation sent by one or more requests.
Bucket	A data structure that can contain multiple data elements such as LAN packets. The OS driver posts buckets to the IOP driver. A bucket is represented as a region of memory. The IOP driver writes incoming packets into this region. The entire bucket is returned to the OS driver at once, regardless of the number of received packets written into it. It might contain 0 or many packets.
Configuring	Setting up the software and hardware, including: Adapter Hardware Configuration - Assigning physical addresses to adapters and physical devices. The IOP is such a device. DDM Configuration - Setting options that control how the DDM utilizes its resources. IOP Configuration - Setting options that control how the IOP utilizes its resources. IOP Driver Configuration - Matching uncontrolled adapters and devices to DDMs that can control them, and sending messages assigning the adapter of device to the selected DDM(s). System Configuration - Assigning uncontrolled adapters to IOPs that can control them.
Configuration Dialogue	Messages between a module and the system that (1) allow the system to display a configuration screen, prompt the user for input, and return that input to the module, and (2) support reading drivers from and writing them to a floppy disk.
Configuration Service	A dialogue for installing or upgrading modules and initializing or modifying configuration parameters.
Device	An I/O object that refers to an I/O facility or service. Adapters are the objects of hardware configuration, while logical devices are the objects of software configuration.
Device Driver Module (DDM)	A module that abstracts the service of an I/O device and registers it as an I$_2$O Device. A DDM can be a hardware driver module, an intermediate service module, or both. This specification refers to both types of modules as DDMs, except where the distinction between the two is important. A DDM registers I$_2$O devices of various classes as appropriate.
Driver Installation	Importing a new or upgraded driver to an IOP. This function adds to or replaces a current driver in the I/O platform's DDM store.
Driver Load	Adding a driver to the IOP operating environment. This function generally copies a driver into the operating environment and then causes the module to initialize.
Embedded I/O Platform	*See I/O Platform*.
Embedded I/O Processor (EP)	The processing element of an I/O platform.

Term	Description
Embedded Kernel Layer	The layer that abstracts OS kernel services for the device driver module (see I_2O Real Time OS).
Event Handler	The service routine that processes the events, including messages addressed to a particular Message Handler.
Execution Environment	The environment within an I/O platform in which a device driver module executes.
External Connection Table (XCT)	A list of connections between DDMs across IOPs.
Hardware Device Module (HDM)	A module that understands and controls a physical device or adapter interface. An HDM is a type of device driver module (DDM). This specification uses the more general term DDM, except where the distinction between HDMs and intermediate service modules is important.
Hardware Resource Table (HRT)	A list of adapters (including sockets or slots that can contain adapters) and their configuration information, including the identity of the controlling HDM. This table tells the host of any adapters the IOP controls, and thus that the host should not touch, as well as adapters the IOP can control.
Hidden Adapter	An adapter that is physically invisible to the host. This adapter is always assigned to an IOP. The host cannot configure this adapter.
Host Node	A node composed of one or more application processors and their associated resources. Host nodes execute a single homogeneous operating system and are dedicated to processing applications. The host node is responsible for configuring and initializing the IOP into the system.
Host Operating System (Host or OS)	The control program executing on the host. This may be the BIOS code, the host bootstrap code, or the final operating system for application programs. Also called the host or OS.
I_2O Device	An instance of an I/O service provided by a DDM.
I_2O Real-Time OS (IRTOS)	A special-purpose real-time OS designed to support high-speed, low-overhead I/O operations.
I_2O Sub-system	One or more IOPs in a single machine controlling any number of adapters and their devices. All IOPs share the same system environment and are treated as peers.
I/O Platform (IOP)	A platform consisting of a processor, memory, I/O adapters, and I/O devices. They are managed independent from other processors within the system, solely for processing I/O transactions. Also called *embedded I/O platform* or *embedded I/O processing node*.
Inbound Queue	A message queue of a particular I/O platform that receives messages from any sender (host or another IOP).
Installation	Transferring a software component into IOP permanent store.
Intermediate Service Module (ISM)	A stacked module that sits between an OS service module and a hardware device module, providing some specialized function. An ISM is a type of device driver module (DDM), and this specification uses the more general term DDM, except where the distinction between ISMs and hardware device modules is important.
Load	Transferring a software module into executable memory.

Term	Description
Logical Configuration Table (LCT)	A list of logical devices whose services are abstracted by the I/O platform (through a device driver module). The host and other IOPs query this table about available resources.
Logical Device (registered I_2O device)	An individual function whose service is abstracted by a device driver module. No direct correlation exists between physical devices and logical devices. That is, one physical device can provide multiple logical devices (example: a four-port Ethernet card) or, multiple physical devices can provide a single logical device (such as RAID). An intermediate service module can combine multiple logical devices into another logical device, such as a RAID (block storage) ISM. Alternately, the module can divide a physical or logical device into multiple logical devices, such as a Fibre Channel port with LAN and SCSI emulation.
Message Handler	The service routine that processes received messages addressed to a particular I_2O device.
Messaging Layer	The messaging layer provides the communication and queuing model between service modules. The messages passed are in an OS-neutral format.
MessengerInstance	The messaging layer running on a particular platform, initializing, configuring, and operating its client modules. Each processor or SMP group has a single MessengerInstance. Each IOP has a MessengerInstance.
Module Parameter Block	A structure that contains the configuration information for a module and the devices it controls. The size and structure of the data is defined by the device driver module. The parameter block is saved by the I/O platform. The module parameter block is supplied to the device driver module when it is initialized.
Module Header	Each DDM contains a module header providing generic information that identifies the driver's abilities. Using the header, the IOP determines which DDMs to load and which adapters and devices it assigns to the DDM.
OS Service Module (OSM)	A driver module that interfaces the host OS to the I_2O message layer. In the split driver model, the OSM represents the portion of the driver that interfaces to host-specific APIs, translating them to a neutral message-based format that goes to a hardware driver module for processing.
Outbound Queue	A message queue for a specific I/O platform for posting messages to the local host, in lieu of the host's Inbound Queue.
Peer Operation	Sending messages between modules on different IOPs.
Peer-to-Peer Operation	Transporting data and control structures between modules on different IOPs.
Physical Device	A set of hardware that operates independent of any other physical device. A physical device is controlled by either exactly zero or exactly one hardware driver module.
Physical Location	A set of attributes that uniquely identify an adapter or a physical device, such as the bus or slot number, I/O or memory address, or the programmed device number used to control the device.
reserved	When a field is reserved, its value is set to zero when created, and ignored on reception.
System Configuration Table (SysTab)	A table built by the host that informs the I/O platform of the existence and addresses of other IOPs.
Target ID (TID)	Logical address of a service registered with the message layer. The target ID is the address the message layer uses to deliver requests to a service module.
TBD	To be determined.

Term	Description
Transport Layer	The abstraction of DMA and access to adapters.
Unassigned Adapter	An adapter not assigned to or controlled by an IOP.
Visible Adapter	An adapter the host can see, independent of whether the adapter is assigned to an IOP. The host performs the bus configuration of these adapters.

1.8 Conventions

This document uses the following text, numeric, and interface conventions.

1.8.1 Text Conventions

Table 1-6. Text Conventions

Text	Description	Examples
italics	variable, mnemonic, equated value, or first reference to an I$_2$O term	*n, STATUS_OK, TRUE, Logical Configuration Table*
ALL_CAPS	mnemonic, equated value	*STATUS_OK, TRUE, FALSE,*
Helvetica	I$_2$O field name or structure name	LCT, TargetAddress
Bold Italics	I$_2$O message names	***ExecIopReset***
Bold w/()	IRTOS API functions	**i2oDdmCreate()**

1.8.2 Numeric Conventions

Numbers in this document are represented in three formats: binary, decimal, and hexadecimal, as illustrated in Table 1-7.

Table 1-7. Numeric Conventions

Type	Examples	Description
Binary	0110b	Numeric expression always ends with the suffix b. Valid digit values are 0, 1, and x. The symbol x indicates a do not care.
Decimal	21 64K 10,562	No suffix except for multipliers (K=1024; M=1,048,576; G=1,073,741,824). Valid digit values are 0 through 9. For readability, a comma (,) every third digit from the right separates long numbers.
Hexadecimal	0C40h 00-C3-5Fh 0C00-0000h	Numeric expression ends with the suffix h. Valid values for a digit are 0 through 9, A through F, and x. The symbol x indicates a do not care. For readability, a dash (-) every second or fourth digit from the right separates long numbers.

1.8.3 Message Naming Conventions

{class}{noun}{verb}

where:

class is the abbreviation of the message class

noun is the abbreviation or name of the object

verb is the service to be performed on that object

Example:

DdmAdapterAttach

Ddm identifies a DDM class message

Adapter identifies the object of the operation

Attach describes the operation on the adapter

1.8.4 Interface Conventions

result = **i2o{noun}{verb}(**Parameters**)**

where:

noun is the abbreviation for the name of the object

verb is the function to perform on that object

Example:

void = **i2oFrameSend (**pFrame, &status**)**

Frame identifies an operation on the message frame

Send describes the operation on the message frame

1.9 Related Documents

This specification refers to the following documents by their codes:

[PCI] **PCI Local Bus Specification**, Revision 2.1, June 1, 1995, PCI Special Interest Group.

[SCSI_2] **SCSI-2** ANSI X3.131-1994, ISO/IEC 9316-1:1994.

[SCSI-3] **SCSI-3 Primary Command Standards** (SPC) X3T10/0995D.

1.10 Acknowledgments

This specification represents the collaboration of The I_2O Special Interest Group, which includes representatives from the following companies:

Steering Committee:

- 3Com Corporation
- Compaq Computer Corporation
- Hewlett-Packard Company
- Intel Corporation
- Microsoft Corporation
- NetFRAME Systems Inc.
- Novell, Inc.
- Symbios Logic Inc.

Member Companies:

- Acer America Corporation
- Adaptec
- Advanced Risc Machines Ltd (ARM Ltd.)
- Advanced Telecommunications Modules, Ltd.
- AMCC
- Amdahl, C.G.
- American Megatrends, Inc.
- Annabooks
- Applied Microsystems Corporation
- Asante Technologies, Inc.
- AST Research, Inc.
- Auspex Systems, Inc.
- Award Software International, Inc.
- BMC Software
- Bytestream Data Systems
- Cabletron Systems, Inc.
- California Polytechnic State University
- Cheyenne Software
- Cyclone Microsystems, Inc.
- Data General Corporation
- Dell Computer Corporation
- Digital Equipment Corporation
- Distributed Processing Technology
- Octopus/Qualix Technologies
- Olicom A/S
- Olivetti Advanced Technology Center, Inc.
- Phoenix Technologies, Ltd.
- PKWARE Inc.
- PLX Technology, Inc.
- QLogic Corp.
- RAMix Inc.
- Raptor Systems, Inc.
- SAIC Ideas Group
- Samsung Electronics Co., Ltd.
- Seagate Software
- Serano Systems Corp.
- Siemens Nixdorf
- Simpact, Inc.
- SMC
- Soliton Systems, KK
- Storage Dimensions, Inc.
- Storage Technology Corporation
- Super Micro Computer, Inc.
- Syred Data Systems
- SysKonnect
- Tandem Computers Incorporated
- Target Technologies, Inc.

- Dolphin Interconnect Solutions, Inc.
- Emulex Corporation
- Force Computers, Inc.
- Fujitsu, Ltd.
- Galileo Technology Incorporated
- Harris & Jeffries, Inc.
- ICP vortex Computersysteme GmbH Adaptec, Inc.
- Industrial Technology Research Institute, CCL
- Initio Corporation
- Integrated Systems, Inc.
- Intergraph Corporation
- Interphase Corporation
- Madge Networks, Ltd.
- Matrox Electronic Systems Ltd.
- Teknor Industrial Computers, Inc.
- Tekram Technology Co., Ltd.
- Telematics International
- Texas Microsystems Inc.
- The Santa Cruz Operation, Inc.
- Tiva Microcomputer Corp.(TMC)
- Topmax Corporation
- Tricord Systems, Inc,
- Tundra Semiconductor Corporation
- Tyan Computer Corp.
- Unisys Corporation
- V3 Semiconductor
- Veritas Software
- Western Digital Corporation
- Wind River Systems
- Xpoint Technologies, Inc.
- Znyx Corporation

2
Technical Overview

This chapter presents a technical overview of the I_2O system. It assumes that you have read the conceptual overview in Chapter 1.

2.1 I_2O System Technical Overview

The intelligent I/O architecture defines an environment for creating device drivers that are functionally divided between the host operating system and an intelligent I/O subsystem. The communication model described in this specification is a message-passing protocol analogous to a connection-oriented networking layer. The transport layer provides a hardware abstraction of the I/O infrastructure to the architecture, leaving the message layer independent of the system hardware.

This technical overview describes the device driver and explains how it is split to achieve platform independence. It then describes the messaging layer that enables splitting the driver across platforms. From the device driver viewpoint, two new interfaces are introduced:

1. the interface produced by splitting the driver
2. the interface to the *messaging layer* that provides the communication service between the split components

2.1.1 Hardware Architecture

The I_2O operation is optimized for a single *host* and an intelligent I/O subsystem containing a number of I/O processors. A host is one or more application processors and their resources, executing a single homogeneous operating system. Figure 2-1 shows a typical hardware architecture, with a host entity and multiple embedded I/O processor entities. In this document, an I/O processor entity is called an *I/O platform* (IOP), and is dedicated to processing I/O transactions. It consists of a processor, memory, and I/O devices.

The I_2O specification supports many variations, but the following are the two most common implementations:

1. the I/O processor on the motherboard designed to control both I/O adapters on the system bus and embedded I/O devices (including I/O adapters on its own expansion bus).
2. the I/O processor on a feature card (such as a LAN controller) designed to control its own private functions.

A *system I/O adapter* is visible from the system bus and thus might be controlled by any IOP or the host. The adapter can be built into the motherboard, or it can be an add-in feature card. Classically, system I/O adapters are usually controlled by a driver executing entirely on the host, so special provisions are required for an IOP to control a system I/O adapter. That is, not all system I/O adapters can be controlled by an IOP. An example is that the interrupt signal from the adapter must be routed to the IOP instead of the host.

A *private I/O adapter* is bundled with an I/O processor's local system (such as I/O Platform 2 in Figure 2-1) and its driver must execute on that processor. Because the host cannot directly access the adapter, it is said to be *hidden*. The I/O platform and its private adapters can be on a typical add-in feature card, or the I/O platform can have one or more expansion buses of its own, thus accommodating additional feature cards.

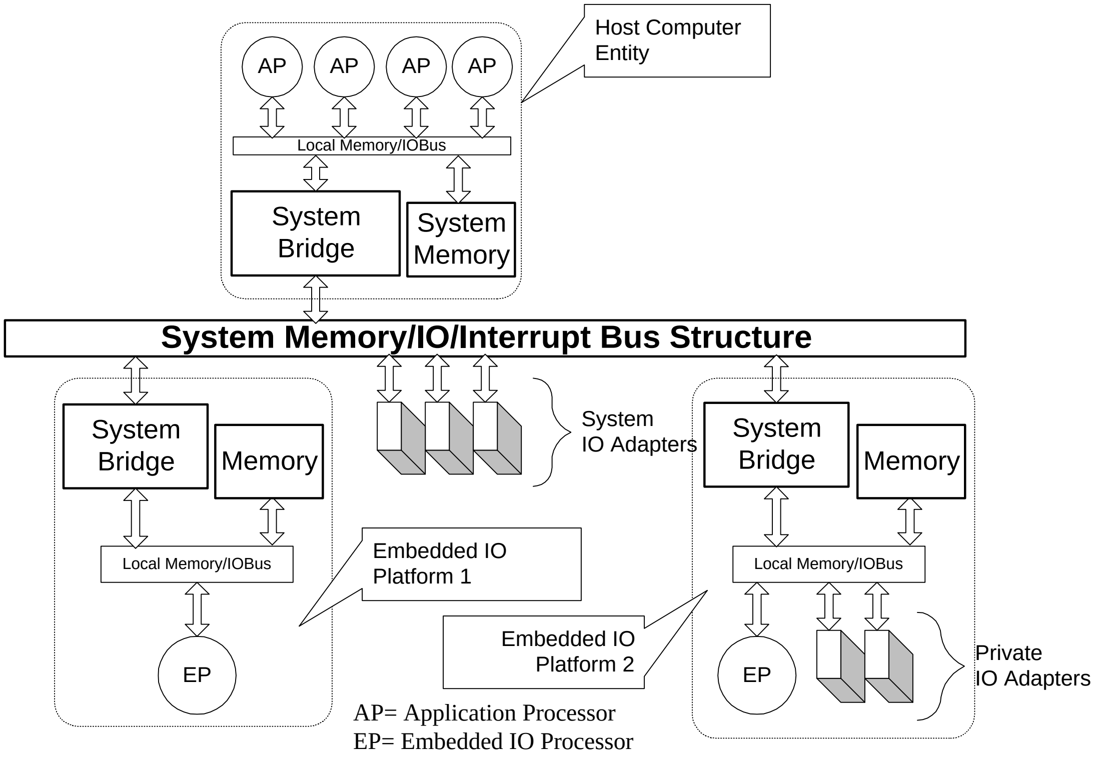


Figure 2-1. Intelligent I/O Hardware Architecture

2.1.2 Split Driver Model

Splitting the device driver in the class-specific region and defining a standard message-passing interface between the two resulting modules means that they can be physically separate. The modules can execute on different processors and even in different operating environments. This *split driver model* is illustrated in Figure 2-2.

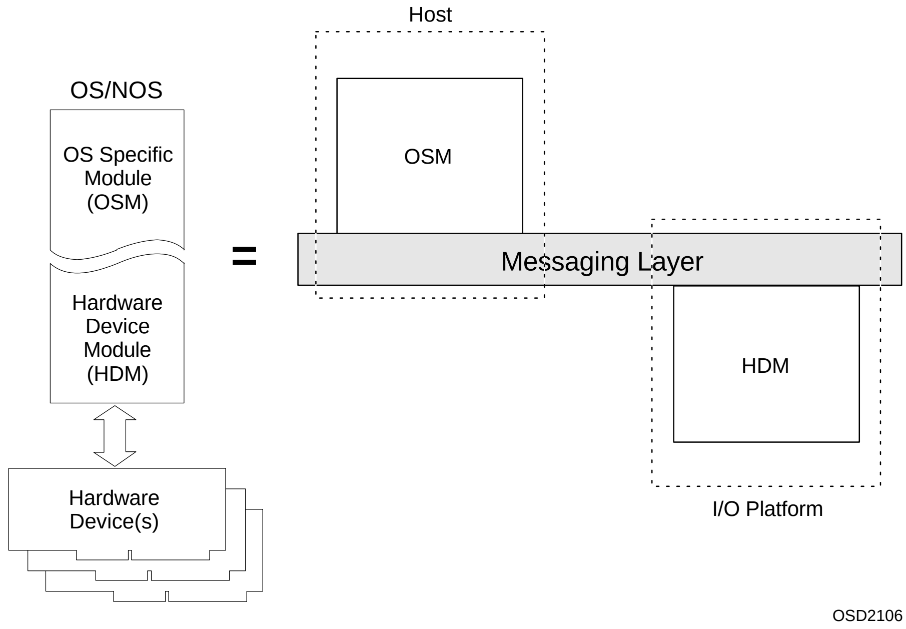


Figure 2-2. I$_2$O Split Driver Model

Splitting the driver as shown in the figure produces two modules:

1. ***OS-specific module* (OSM).** The upper module provides the interface to the operating system. Typically, the OS vendor supplies this module, which contains no hardware-specific code.
2. ***Hardware device module* (HDM).** The lower module provides the interface to the I/O adapter and its devices. The hardware vendor supplies this module, which contains no OS-specific code.

Besides these two basic module types, this document uses two other terms for modules:

1. ***Intermediate service module* (ISM).** Splitting the device driver more than once, or adding functionality between the OSM and HDM creates stackable drivers. This puts one or more of these intermediate modules between the OSM and HDM. An independent software vendor can supply ISMs.
2. ***Device driver module* (DDM).** HDMs and ISMs are often referred to collectively as device driver modules or DDMs, because, in many aspects, their behavior is identical. This is especially true from the viewpoint of the host OS. This specification uses the general term DDM, unless it needs to distinguish between HDMs and ISMs.

For any class of I/O operation, only one version of an OSM is necessary for multiple DDMs of that same I/O class. The OSM need not be preconfigured for a specific vendor's DDM. An OSM can locate and connect with each DDM using the facilities of the messaging layer.

2.1.3 Messaging Layer Architecture

Splitting the driver requires message definitions of the new interface between those modules. The I$_2$O communication layer is the *messaging layer*, which delivers I/O transaction messages from one software module to another, anywhere in the I$_2$O domain. The layer provides:

- message delivery between modules
- configuration registry
- abstraction from the physical platform

The messaging layer is a network of MessengerInstances, as illustrated in Figure 2-3. A *MessengerInstance* is a collection of services that support initializing, configuring, and operating its client modules. Each MessengerInstance is the messaging layer running on a single platform; there is one instance per processor or Symmetric MultiProcessor group.

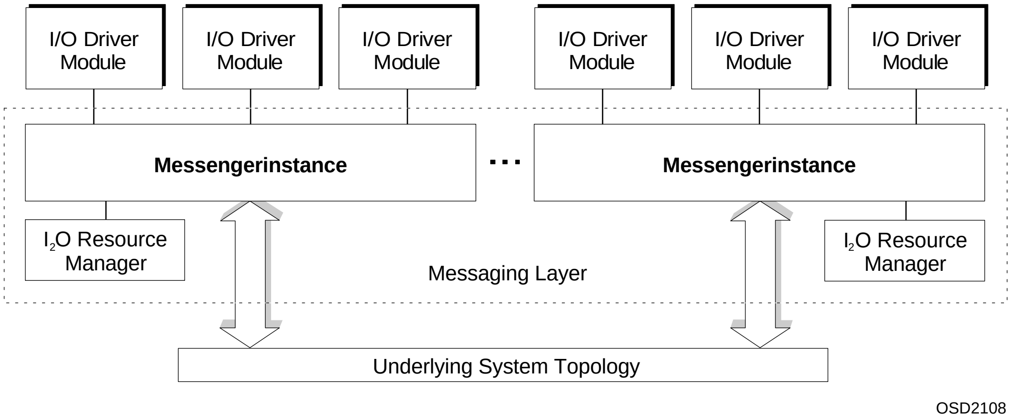


Figure 2-3. Communication Service Model

The messaging layer provides the communication service for one module to send messages to another. This postal manager is referred to as the *messenger*. The messaging layer only transports the messages, and provides no I/O functionality. As illustrated by the split driver communication model in Figure 2-2, the HDM (as the lower layer of the driver stack) provides service to the OSM (the upper layer of the driver stack). The messenger is not concerned with this hierarchical connection between modules, however. From the messenger's point of view, the messaging layer provides the same service for the HDM, ISM, and OSM, so these modules are considered peers. As far as the message service is concerned, the messenger is the service provider, and the HDM, ISM, and OSM, also referred to as *service modules*, are the service users. Figure 2-4 illustrates the overall communication architecture correlating to the hardware architecture in Figure 2-1.

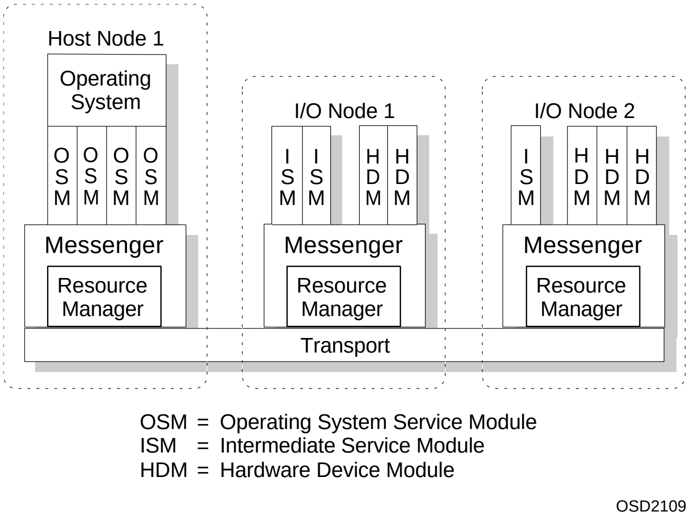


Figure 2-4. Communication Architecture

The messaging layer provides the configuration registry that enables modules to locate and configure with other modules. Resource management is distributed among the messengers. An executive resource manager in the host messaging layer oversees peer-to-peer connections between IOPs.

For DDMs, the messaging layer provides the abstraction and adaptation between the DDM and all of the system's environment (RTOS, host OS, bus structure, and so forth). The only interface between modules is through the messaging layer. The interface is a set of service APIs that provide all the services for the module:

- *transport services* for accessing system memory, moving large blocks of data between modules, and accessing I/O adapters;
- *message services* for locating and communicating with other modules;
- *RTOS services* for the execution environment, including memory allocation, task management, and so forth.

Figure 2-5 shows a more detailed model of the MessengerInstance.

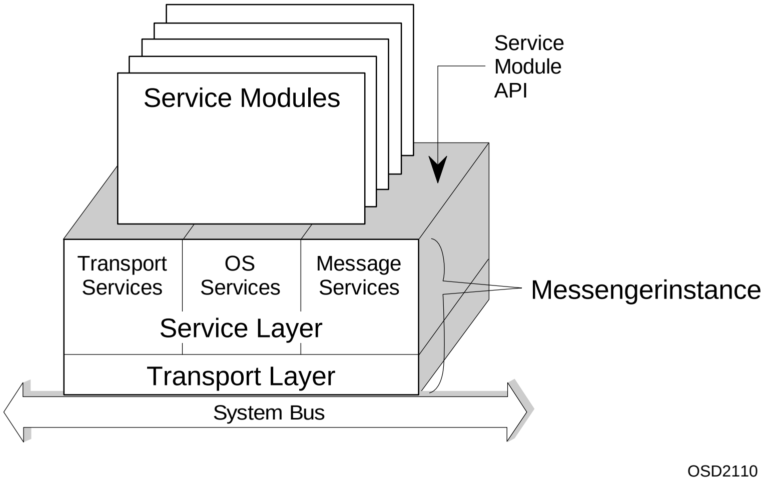


Figure 2-5. Message Service Model

2.1.3.1 MessengerInstance Architecture

This document describes an open architecture for developing a communication platform, the MessengerInstance. The architecture is independent of the operating system, processor platform, and system I/O bus. This version of the specification defines a transport interface between MessengerInstances for a shared memory environment, but does not preclude defining other transport environments in future revisions.

The transport sublayer abstracts the system bus and thus provides the interface to the service modules for accessing system resources. This mechanism transfers data between a module and its I/O devices or system memory, and between modules. It also provides access to control registers of devices residing on the system and expansion buses.

The messaging layer operation is independent of the module's class of operation and the operating environment of other platforms. A system includes:

- a host MessengerInstance
- a number of MessengerInstances abstracting I/O channels. These are called IOP MessengerInstances, because each is the messaging layer on an IOP.

When this document uses the term *MessengerInstance* without clarification, it implies both host and IOP MessengerInstances.

From a system perspective, the host is the primary user of the I/O services, and the modules on the IOPs are the service providers. This specification also allows peer-to-peer operation between modules on different IOPs.

2.1.3.2 Message System Interfaces

The physical and logical interfaces presented in the I_2O environment are illustrated in Figure 2-6. This document specifies the I_2O core interface between an IOP and a loadable DDM, and the shell interface between either two IOPs, or the host and an IOP. It also includes message-based interfaces between an OSM and a DDM, or between two DDMs. The specification also includes the configuration interface, where the IOP interacts with the system, and ultimately, the user, during configuration and driver installation.

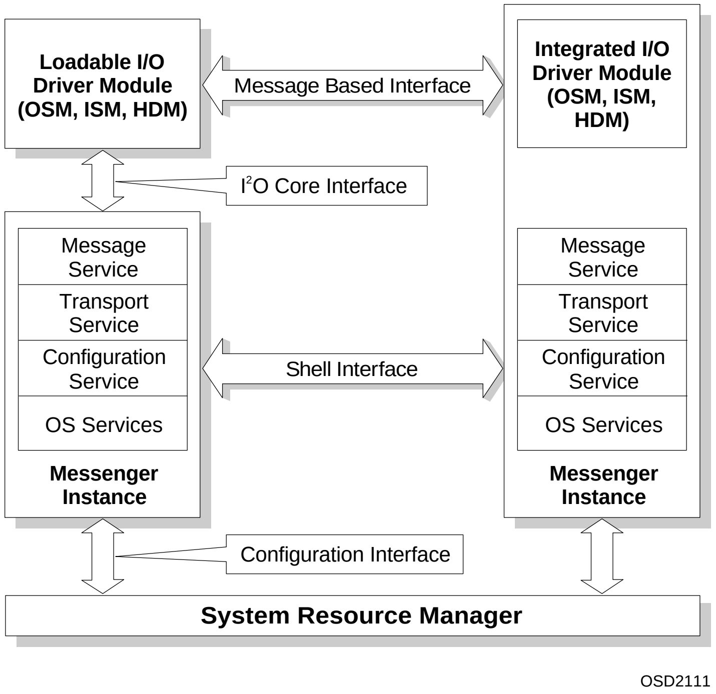


Figure 2-6. I$_2$O Interface Topology

2.1.3.2.1 Core Interface (Between IOP and DDM)

The interface between an I/O module (HDM or ISM) and the IOP MessengerInstance is the I$_2$O *core interface*. This service interface is an operating platform for DDMs, including:

- an API, consisting of function calls
- a message-based interface where the I/O driver and the MessengerInstance exchange messages to configure and manage the modules

This mechanism configures and initializes the DDMs, registers the I/O devices it controls, and establishes a path to other modules for exchanging messages.

The core specification in Chapter 5 is particularly relevant to vendors writing device driver modules for non-intelligent adapters that load onto a generic I/O platform. It provides for a standard interface between the DDMs and the IOP. In this case, the MessengerInstance is incorporated in an I$_2$O real time operating system named *IRTOS*. The specification details how the I/O platform provides the operating environment for its loadable device driver modules: abstracting the system and I/O buses and the underlying operating system, and configuration and registration management. This involves a collection of services, each of which represents an abstraction of a component common to all drivers:

- **Configuration services:** The interface with the resource manager
  - loading DDMs
  - software initialization
  - hardware configuration
  - registering I_2O devices
- **OS services:** The interface with the embedded kernel
  - task scheduling
  - memory allocation
  - interrupt service
  - timer service
- **Message services:** The interface with other modules
  - connection
  - message delivery
- **Transport services:** The abstract system bus, for moving control information and data across bus topologies through a set of interfaces
  - adapter access: an abstraction of the system memory and I/O bus.
  - system memory access: a translation of memory references between system and direct references.
  - data transfer: an abstraction of DMA capabilities.

See Chapter 5, *I_2O Core Specification*.

2.1.3.2.2 Message-based Interfaces Between Service Modules

When OSMs and DDMs exchange messages, they use a *message-based interface*.

Each I_2O class has its own message-based interface designated by one of the *message class specifications*. Each class includes messages and a protocol for replying to them. A set of *utility messages* is common to all message classes. The messages specific to a particular message class are called *base class messages*. For value-added functionality, this specification also supports messages that extend the base class. These messages are considered *private*. They are defined by individual organizations and are not included in the I_2O specification. The defining organization can keep the private extensions strictly internal, or it can choose to publish part or all of them. As part of the I_2O specification, each module identifies the extensions it supports.

Each class specification is completely independent of the operation of the messaging layer, because the messenger is involved only in delivering messages. A MessengerInstance does not require any knowledge of class specifications.

See Chapter 6, *Class Specifications*.

2.1.3.2.3 Shell Interface

Message delivery between platforms requires the *I_2O shell interface* between those platforms. This system interface contains:

- a register-level hardware interface for sending and receiving messages
- executive message definitions, and a protocol for exchanging those messages.

The physical portion of the shell interface specifies a single queuing model for shared memory architectures. This queuing technique for transferring messages uses:

- One *inbound queue* for each IOP. The inbound queue of a platform receives messages from all other platforms, including the host.
- One *outbound queue* for each IOP. The outbound queue of all IOPs collectively functions as the input queue for the host. This allows each I/O platform to provide hardware support for efficiently passing messages without requiring additional host hardware.

The queuing model is transparent to the operation of the DDMs.

The logical portion of the shell interface is the protocol between MessengerInstances for establishing, maintaining, and releasing message paths. Each MessengerInstance communicates by placing messages in the target's inbound message queue. Thus, the operation of a MessengerInstance is independent of the operating environment of other platforms.

Message protocol is based on a loosely-coupled request-reply pairing. Each I/O class defines its own behavior. The reply behavior is specified by message. The reply can be synchronous or asynchronous with the request, delayed or deferred, and each request can have less or more than one reply.

See Chapter 4, I_2O *Shell Interface Specification* and Chapter 6, *Class Specifications*.

2.1.3.3 System Execution Environment

The OSM executes in a specific host OS environment. The message-based interfaces, discussed in section 2.1.3.2.2, provide direct communication between an OSM and a DDM. The other interfaces, such as that between the OSM and the host MessengerInstance, are host-specific and not part of this specification. The OS provides a single MessengerInstance for initializing the system and providing the message service between its OSMs and any IOPs. In addition, the host's resource manager provides executive control over system resources and establishing connections between IOPs.

The execution environment for the IOP has additional facilities for loading, initializing, and managing DDMs. In general, these extended services are provided by a real-time OS or embedded kernel, abstracted through the API interface specified in Chapter 5, *I_2O Core Specification*. The system execution environment is depicted in Figure 2-7.

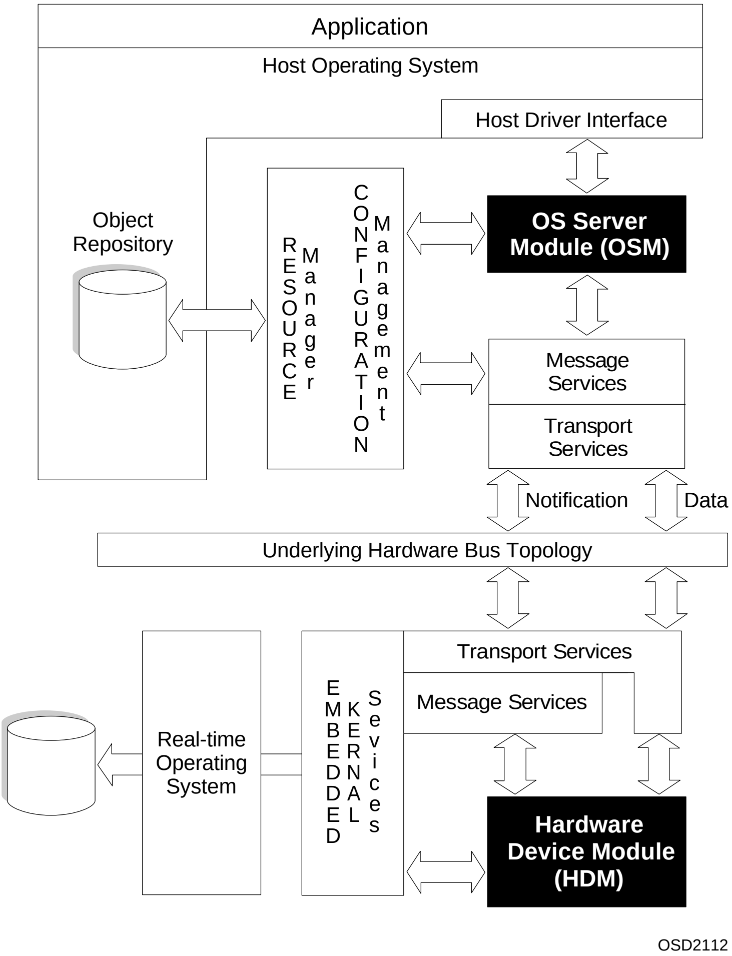


Figure 2-7. I$_2$O Execution Environment

2.1.3.4 System Resource Manager

The host manages system resources. This interface is defined by the *executive message class*, specified in Chapter 4. The resource manager initializes each IOP and provides it with the information about other IOPs. An I$_2$O system descriptor table lists the location of each IOP's inbound message queue. The system resource manager notifies all IOPs when it detects a change in the system configuration.

The system resource manager also provides the *configuration dialogue*, information the IOP or DDM displays on the system console and that prompts the user for input. The host dispatches the configuration dialogue at its own discretion.

2.1.4 Configuring and Initializing

The various operating systems and industry standards offer many configuration facilities. The I_2O configuration facilities do not compete with those systems, but rather provide a standard service interface for controlling the configuration of I_2O components. The configuration model simplifies resource management within the I_2O environment by providing a sort of mezzanine approach. The configuration behaves as a distributed repository of information on available resources, rather than representing a hardware topology of where resources physically reside. The MessengerInstance tracks available resources, their location and attributes, the relationship between modules and physical devices, and so forth.

2.1.4.1 Initialization of the I_2O System

The host must initialize the I_2O system. Each IOP is responsible for its own initialization and prepares its inbound message queue for messages from the host. The host locates each IOP, adds it to the system configuration table, and initializes the IOP’s outbound message queue. The host then provides each IOP with a list of all IOPs and the physical location of their inbound message queue. When an IOP wants to connect to another IOP, it sends the request to the respective IOP’s inbound message queue. The connection request and its reply convey information enabling the two IOPs to establish a direct path for exchanging messages.

2.1.4.2 Configuration of I/O Device Drivers

Each IOP is responsible for configuring its own I/O device drivers.

2.1.4.2.1 Loading DDMs

An IOP must provide methods for storing and loading its own operating environment and DDMs. The IOP also provides storage for module parameter tables used to initialize its devices.

The IOP’s executive service also supports downloading additional drivers from the host.

2.1.4.2.2 Configuration Services

The IOP must keep its configuration persistent from boot to boot. As part of the DDM configuration, the IOP identifies adapters assigned to each DDM. The DDM initializes those adapters. As the DDM brings an adapter on-line, it registers the service that the adapter provides, creating one or more I/O devices. Each I/O device is assigned a message handle or *target identifier* (TID), which is the logical address of the service. To use that service, an OSM must send requests addressed to that TID. The device is listed in the IOP’s logical configuration table. An OSM queries this table to learn which TID controls the particular resource/device it wants to use. No restrictions are placed on the location of the adapters or devices that a module controls.

The configuration service also provides extended API calls for dynamically configuring and replacing adapters, media, and I/O driver modules (see the configuration management messages in Chapter 4.) This version of the specification does not accommodate dynamic upgrading from one version of a DDM to another, but it does prevent installing or updating a driver that corrupts operation.

Any time a new or replacement driver is installed on an I/O platform, it is tagged *experimental*. The old driver is tagged *obsolete* and retained until the operation of the new driver is confirmed.

The next time the IOP is booted, it loads the experimental version of the driver, changes the experimental status to *suspect*, and waits for the host to send a configuration validation message. If the IOP does not receive a confirmation within a reasonable period, it may invoke a configuration dialogue asking the user to accept, reject, or defer the suspect driver. If the user accepts the new (suspect) version, the old (obsolete) version is removed from the IOP's store and the suspect status of the new driver is cleared. If the user rejects the suspect version, it is removed from the IOP's store, and the obsolete tag on the original version is cleared. If the IOP boots a second time and the user neither accepts nor rejects the suspect module, their inaction constitutes an implicit rejection. The suspect version is removed and the old version reinstated.

The initialization mechanism provides the basic primitives for binding IOPs to the host. Another mechanism is used for loading DDMs. Neither mechanism enforces ordering; this allows the initialization sequence to blend with that of the primary host OS.

The I_2O environment does not require that the boot device be I_2O compliant. Boot time I/O support can occur through a BIOS or IPL component that understands the basic underlying hardware architecture.

The system can bootstrap using one of two models. First, any legacy boot model in use today works with I_2O. The second model is based on providing an I_2O service layer that allows I_2O-enabled systems to boot, using either I_2O block storage devices or I_2O remote boot devices. The class definition for remote boot devices is not defined at this time.

2.1.5 I_2O Environment

An I_2O system contains one or more I_2O *segments* bound by an I_2O bridging or a routing agent, as illustrated in Figure 2-8. Operation of bridging and routing agents is transparent to the operation of the DDMs, OSMs, and IOPs. Bridging provides I_2O communication between coupled platforms, that is, platforms sharing the same contiguous system bus architecture and coherent memory. Routing provides communication between discontiguous platforms. Bridging primarily involves connection setup and allows an IOP to send messages directly to an IOP in another segment. Routing usually involves store and forward techniques translating between the address domains. The definition of the bridging and routing agents is outside the scope of this version of the specification. This document specifies operation solely within an I_2O segment.

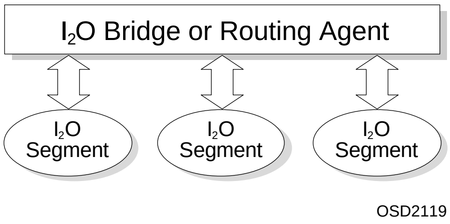


Figure 2-8. I_2O System Topology

2.1.5.1 Shared Memory Model

Each I_2O segment contains a host and one or more IOPs. The host configures and initializes the components within the segment. On each host resides a system resource manager. Figure 2-9 shows how access is shared among the components in an intelligent I/O segment.

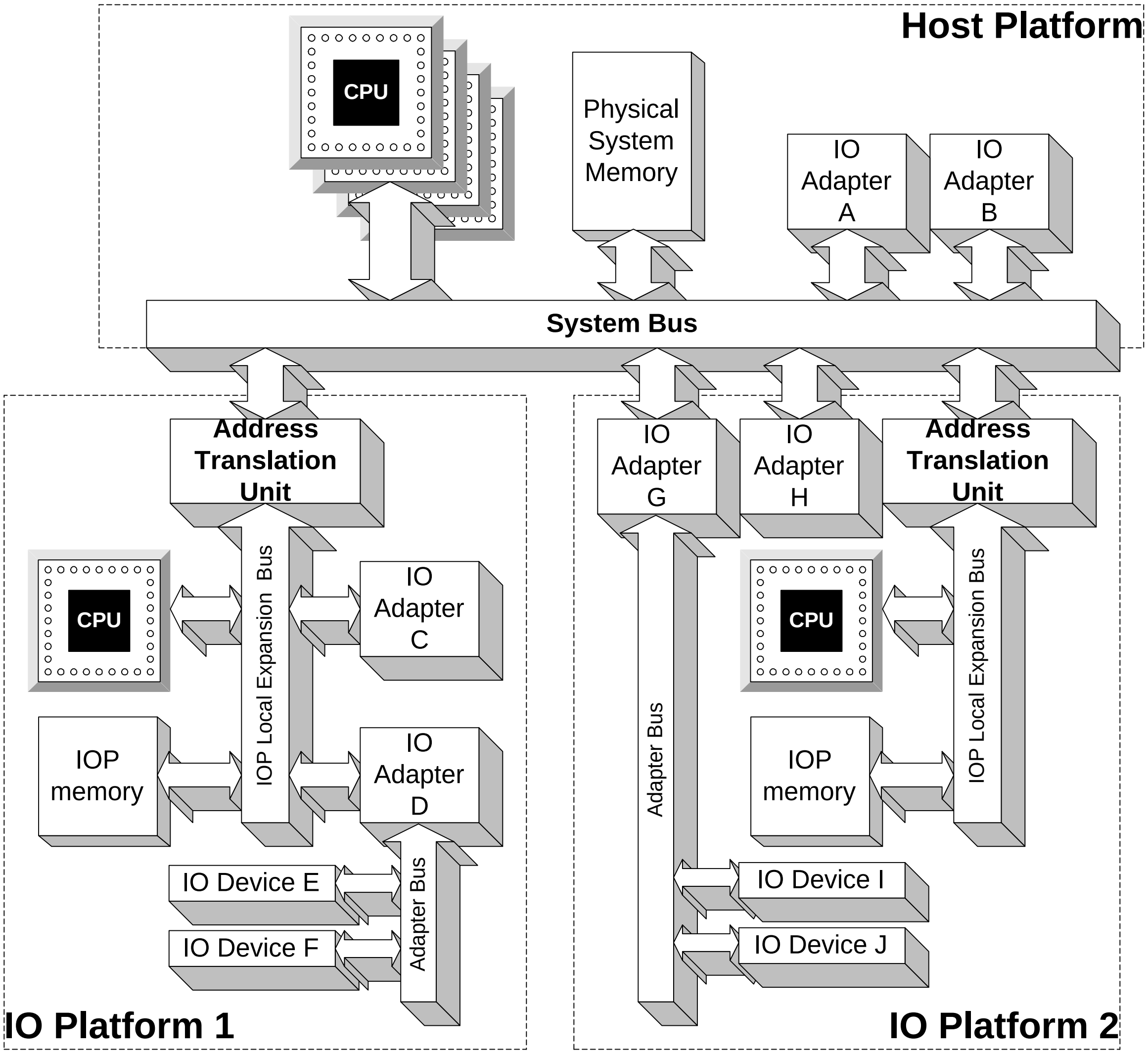


Figure 2-9. I_2O Segment Example

2.1.5.2 I/O Device Domains

Figure 2-10 illustrates the domains created by the example in Figure 2-9. I/O adapters A and B are directly controlled by the host OS, while I/O adapters G and H are on the system bus and controlled by I/O Platform 2. These four adapters are directly addressable from the system bus. I/O adapters C and D, on the other hand, are directly accessed through IOP 1's local bus and are thus private to IOP 1. I/O adapters D and G contain devices accessible through their own bus (for example, a SCSI adapter with SCSI devices).

I/O adapters A and B are controlled by the host OS and do not show up in the I_2O configuration table. All other adapters are controlled by an IOP and appear as an abstracted service (i.e., an I_2O device), as illustrated in Figure 2-10. A device appearing behind an IOP always appears abstracted. The best example of this is the SCSI controller. The HDM for a SCSI controller detects and registers devices on the SCSI bus. Those devices are accessible only through messages sent to the SCSI HDM.

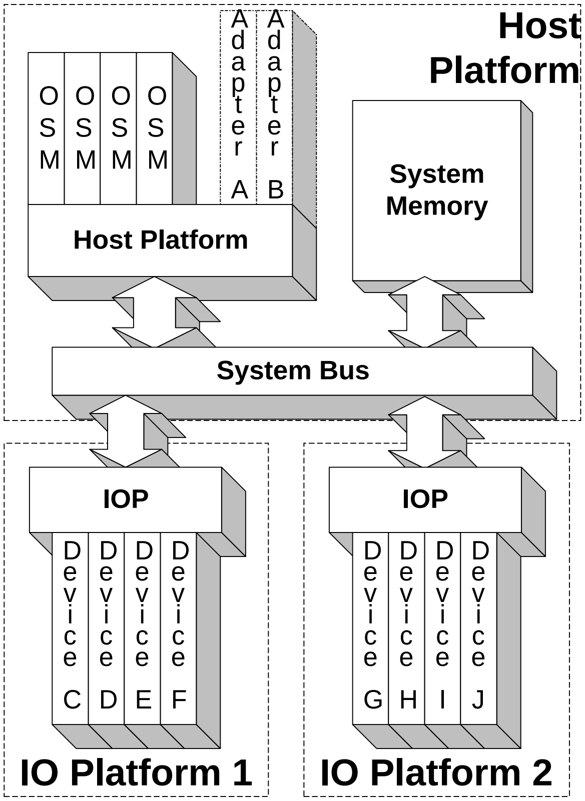


Figure 2-10. Abstracted View

2.1.5.3 Accessing Adapters

Even though the outcome is a set of abstracted services (shown in Figure 2-10), Figure 2-9 illustrates adapters accessed directly within the IOP's domain (like adapters C and D) and others on the system bus (like adapters G and H). In addition, an IOP can contain secondary I/O buses where adapters can reside. The IOP must enable a device driver to access the adapter by providing a transport layer that abstracts system and expansion bus access.

The IOP provides a generic interface for DDMs by abstracting all its transport capability. That is, all transport services for accessing an adapter are generic API functions, independent of the physical topology. This interface supports I/O transactions, memory transactions, and configuration access.

2.1.5.4 Address Domains

The system is typically composed of the host and a number of IOPs bound together by the system memory-I/O infrastructure. Each IOP contains and manages its own local memory-I/O system, independent of the others.

The host and an IOP view memory differently: The host sees system memory and memory of memory-mapped adapters as a single address domain. On the other hand, an IOP's address domain is its own local bus and does not access system memory directly. An IOP must have some local memory mapped into the system domain so that the host (or any other bus master adapter) can directly access that memory via the system bus. This is called *physically shared memory.*

The memory available to each IOP falls into one of three categories, depending on how the memory can be accessed:

1. **System memory**. This memory can be accessed only via the system bus, and thus a memory location can be specified only by a system memory address reference. The IOP utilizes a DMA mechanism to move data between system memory and its local memory.
2. **IOP private memory**. This memory can be accessed only via the IOP's local bus, and thus a memory location can be specified only by a local memory address reference.
3. **Shared memory**. This portion of an IOP's local memory can be accessed via either the system bus or the IOP's local bus, and thus a memory location can be specified by both a system memory address and a local memory address.

Shared memory relative to one IOP is only system memory relative to another IOP, and thus both shared and system memory are accessible to other IOPs. This is not true for private memory, which is available only to the local IOP. Since modules operate in the IOP's local address space, each IOP must be able to translate the address of a shared memory location from a local reference to a system reference, and vice versa.

2.1.5.5 Address Translation Unit

Shared memory is a portion of the IOP's local memory that the rest of the system accesses by an *address translation unit* (ATU). The ATU maps a portion of the IOP's local memory into the system memory domain; a system memory access to that area translates to a corresponding access cycle to the IOP's local memory (see Figure 2-11). This translation allows each IOP its own address space, independent of the system address space, but still lets the host and other IOPs access some of the IOP's local memory. The portion of the IOP's local memory that can be accessed through the system bus is the IOP's shared memory. The ATU maps the IOP's shared memory into the system address space so that the system views it as part of system memory.

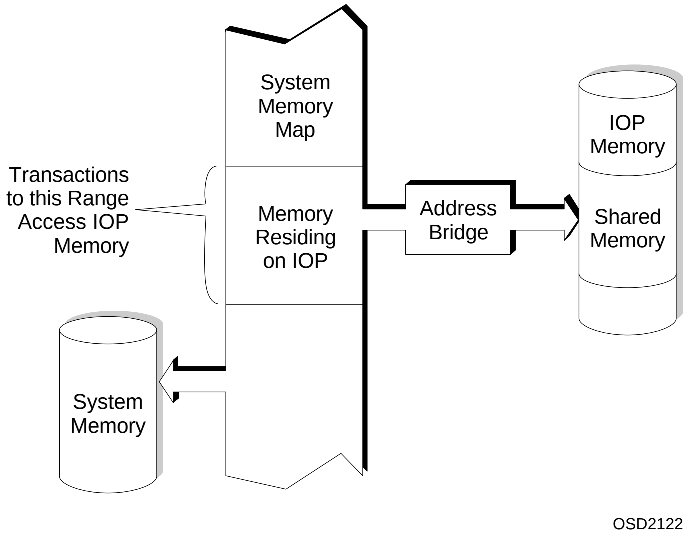


Figure 2-11. Memory Address Translation

The ATU does not map system memory onto the IOP's local bus, because system resources may overwhelm the IOP's memory space. The IOP must supply a transport (DMA) function to move data from its local memory (or local devices) to system memory, and vice versa.

From the system's point of view, shared memory is the portion of an IOP's physical memory that the system can access directly. From the IOP's, HDM's, and ISM's point of view, shared memory is the portion of the local memory that can be accessed by system components outside the IOP.

The address bridge is called an *address translation unit* because the physical address on the system bus must be translated to the physical address on the IOP's local bus. A direct relationship exists, however, between the system and corresponding local bus addresses. For each IOP, the difference between the system and local bus addresses is a constant for all shared memory. This constant is the *base difference* between the IOP's local address and the system address (modulo 2**32). Thus, adding the base difference to the local address of a shared memory location yields its system address. Likewise, subtracting the base difference address from a system address yields the local address. This relationship is valid only for the shared memory on the local IOP.

2.1.5.6 Address Translation

The IOP and DDMs must use the system address, rather than the local address, to refer to a shared memory location when communicating with the host or another IOP. This requires that a module have a translation mechanism that converts a local memory address reference to a system address reference, so it can import and export data from its local memory.

2.1.5.7 Data Shipping

Figure 2-12 shows two methods for delivering data between modules on different IOPs: *pushing* and *pulling*.

- Module A uses pushing to move data in its private local memory to a buffer in module B's shared memory, which is mapped into the system address space. Because this memory is part of module B's local memory, module B now has direct access to the data.
- Module B uses pulling to move data from a buffer in module A's shared memory region, which is mapped into system address space, to its local memory.

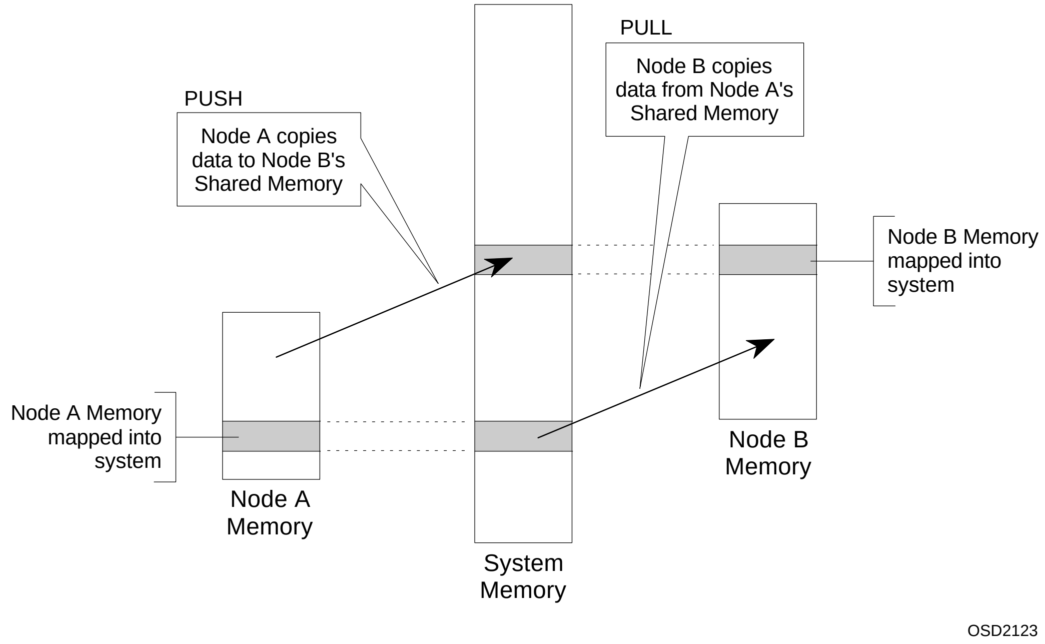

PULL
Node B copies data from Node A's Shared Memory
PUSH
Node A copies data to Node B's Shared Memory
Node B Memory mapped into system
Node A Memory mapped into system
Node B Memory
Node A Memory
System Memory
OSD2123

Figure 2-12. Mechanisms for Transferring Data

In general, pushing memory is more efficient, and preferable to pulling. The message transportation relies on pushing for delivering messages. The IOP allocates message frames in its own shared memory. The host and other IOPs copy messages (push) from their domain into those message frames. For the outbound queue, the host allocates message frames in system memory, and the IOP copies data from its local memory to the system buffer (push). Neither has to read data across the bus.

2.1.5.8 DDM Environment

The DDM's view of the I_2O system is simple. A set of direct services is available to the DDM through the I_2O core service interface. This interface provides the messaging, transport, OS, and configuration services. Additionally, the DDM creates virtual interfaces with other modules through the message-passing protocol. The message service interface enables transporting messages between modules and platforms.

A DDM executes in the IOP's local address domain, but memory references from an OSM refer to system memory, not local memory. When the data is in main system memory, the data pointers (scatter-gather list) in the message frame always use the system address reference. For the DDM to operate on the data, it must translate the system address to a local address by subtracting the base difference from the system address, as explained earlier. In many cases, the DDM must know which system addresses are actually shared local memory and which are not. When the DDM initializes, it learns or calculates the following parameters by calling various API functions:

- Base address of shared memory, the system address where the IOP's memory is mapped
- Length (number of bytes of shared memory), which determines the upper bound
- Base difference between the system and local base addresses

With these parameters, the DDM can determine when a system memory reference is local shared memory, and it can also translate between system and local shared memory references. The IOP also provides API functions to perform these actions for the DDM.

When the DDM controls an adapter on a system bus, it must use the IOP's transport services to access the adapter (read and write to I/O registers and adapter memory). If the adapter is a bus master adapter, it must be programmed with system memory references and not the IOP local bus reference. The DDM must always know whether a memory reference is system or local.

2.1.6 Communication

Initializing the IOP automatically establishes the communication channel between OSMs and DDMs. For peer operation between IOPs, the messaging layer provides the service to establish, use, and tear down communication channels (connections) between modules in the I_2O environment.

2.1.6.1 Opening a Peer Connection

The host initiates a peer connection by sending a message from the host to the IOP, assigning an I/O device registered by a remote IOP to a local DDM (***ExecDeviceAssign*** message). The IOP in turn attaches the I/O device to the local DDM (***DdmDeviceAttach*** message). For this discussion, this local DDM is called the user DDM.

As previously mentioned, each IOP assigns a TID to each local I_2O device as it is registered. That TID becomes the handle for that device. A user DDM uses the TID to address a request to that device. TID assignments are local to an IOP. Before a DDM can send messages to a device on another IOP, a pair of alias TIDs must be established so the requests can be dispatched and their replies routed back to the originator.

1. The local IOP assigns its own alias TID for the remote device. All local DDMs use this alias to specify the remote device.
2. The local IOP creates a connection setup request message supplying the connection information and sends the request to the target IOP's messenger. The request specifies the alias TID. The remote IOP remembers that alias so that, for replies from the remote device to the originating module, the remote IOP can replace the target's TID in the TargetAddress field with that alias.
3. The target IOP validates the connection with the target module and replies to the originating IOP. If the reply is an ACCEPT, it contains another alias TID that the remote IOP assigned to the originating module. The requesting IOP remembers this alias TID so it can replace the originator's TID in the InitiatorAddress when it posts requests to the target device.

An alias TID supports assigning TIDs by the local IOP. DDMs see only TIDs assigned by their local IOP. Since TIDs are not unique within the entire system, the IOP sending the message converts the Initiator Address and Target Address fields in the message frame header from its local assignments to those used in the target IOP. For requests, the IOP replaces the initiator address with the alias TID assigned by the target IOP. It also replaces the target address with the TID originally assigned by the target IOP. Conversely, for replies, the IOP converts the initiator address from the alias TID it assigned to the TID originally assigned by the target IOP; and, it replaces the target address with the alias TID the target IOP assigned.

In other words, the IOP sending a message to a remote IOP substitutes both the InitiatorAddress and TargetAddress. As illustrated in Figure 2-13, the User DDM specifies its own TID and the local IOP's alias for the target device. IRTOS A looks up the TargetAddress and:

(1) determines the target IOP

(2) replaces IOP's alias for the target device in the TargetAddress field with the actual target's TID, and

(3) replaces the InitiatorAddress with the target's IOP's alias for the initiator. Now IOP A posts the message to IOP B's inbound queue. IOP B delivers the message to the target device unchanged.

For the reply path, the target device uses the same TIDs as received. IRTOS B looks up the TargetAddress and:

(1) determines the target IOP

(2) replaces the IOP's alias for the User DDM in the InitiatorAddress field with the actual TID

(3) replaces the TargetAddress with the IOP A's alias for the target. Now IOP B posts the message to IOP A's inbound queue. IOP A delivers the message to the User DDM unchanged.

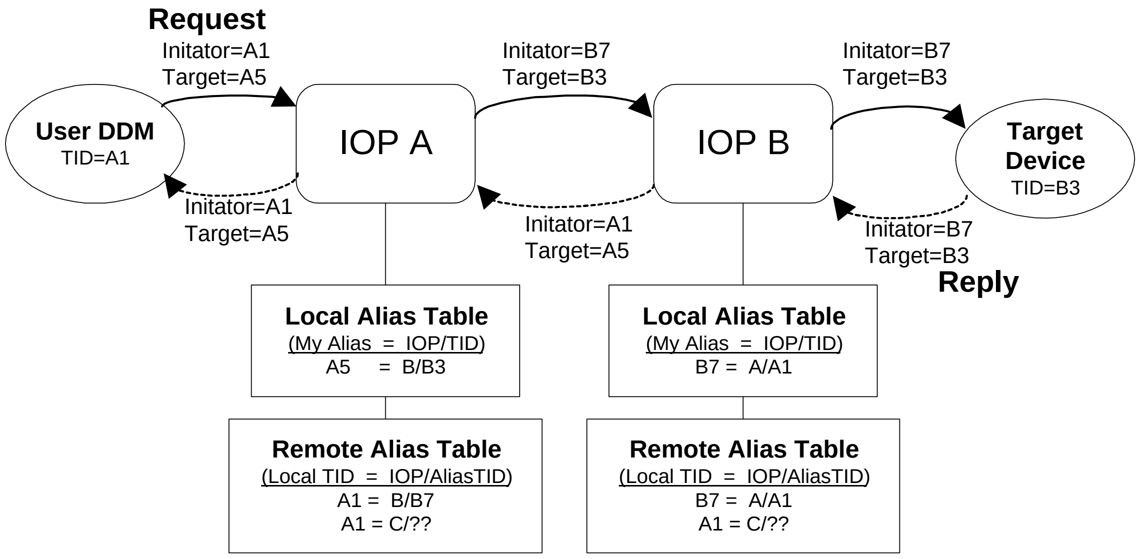


Figure 2-13. Illustration of Peer Message Operation

TID values of 0 and 1 are reserved for the IOP executive and host OSMs, respectively. Since the host dispatches replies based on the initiator context field, and host OSMs are preassigned a unique TID, aliasing is not required between an IOP and the host.

Connecting two modules within the same IOP is a local task. The modules use the TIDs locally assigned. Because alias TIDs are not required, the modules need not formally connect with another local module.

2.1.6.2 Closing a Connection

Because connections are established by assigning alias TIDs, they do not need to be closed between sessions. The only advantage of closing a connection is in recovering the TID for future use and error recovery. (See restrictions on reassigning TIDs in section 2.3.3.5.) A connection can be closed at any time by either end of a connection or by the MessengerInstance. A message informs both modules that the link is shutting down. Both modules must consent to the closure, so either can keep the connection open until all pending requests finish.

2.1.6.3 Sending Messages

Once the IOPs establish a connection, the modules at both ends of the connection can send and receive messages. Messages are sent in an asynchronous fashion and are non-blocking by nature.

2.1.6.3.1 Host Messages

The host uses the following procedure to exchange messages with a DDM:

1. The OSM builds a request and calls the host’s message service, indicating the target IOP.
2. The host MessengerInstance allocates a message frame by reading the target IOP’s inbound message port. This produces the address of a free message frame from the IOP’s free list.
3. The host places the message in the frame, and notifies the target IOP by writing the frame’s address to the IOP’s inbound message port.
4. The IOP inspects the request header’s Target Address field and posts the message to the appropriate DDM.
5. Later, the DDM releases the message frame and the IOP places its address on the free list.
6. If a reply is necessary, the DDM builds one. It uses the initiator address, target address, and the Initiator Context field from the request, and calls the local message service.
7. The IOP’s messenger service, noting that the Initiator Address field contains the value 001h, allocates a message frame from its list of free host message frames. It copies the message into the frame and places the frame’s address in its outbound message queue.
8. The host gets the reply’s message frame’s address by reading the IOP’s outbound message port and dispatches the message based on its Initiator Context field.
9. When the OSM processes the reply and releases the message frame, the host MessengerInstance reallocates it to the IOP by writing its address to the IOP’s outbound message port.

2.1.6.3.2 Peer Messages

The peer operation varies slightly, because of the alias TIDs. The use of the aliases is transparent to the DDMs sending and receiving the messages.

1. The originating DDM builds a request. It places the target's alias TID assigned by the local IOP in the Target Address field and calls the local message service.
2. The sending IOP examines the target address, determines the target IOP and the actual TID for the target DDM assigned by the target IOP, and replaces the target address with the actual TID.
3. The sending IOP looks up the initiator address, finds the corresponding alias TID assigned by the target IOP, and replaces the initiator address with that alias.
4. The sending IOP allocates a message frame by reading the target IOP's inbound message port, and places the message in the frame. It then notifies the target IOP by writing the address of the message frame to the IOP's inbound message port (just as the host did).
5. The target IOP inspects the request message header's Target Address field and posts the message to the appropriate DDM.
6. Later, the DDM releases the message frame and the IOP returns its address to the free list.
7. When a reply is necessary, the remote DDM builds one using the Initiator Address, Target Address, and the Initiator Context field from the request and calls its local message service (exactly the same as with messages from the host).
8. The remote IOP's message service examines the Initiator Address field in the reply and detects that it is not 001h.
9. The sending IOP examines the Initiator Address, determines the originating IOP and the actual TID assigned by the originating IOP, and replaces the Initiator Address with the actual TID.
10. The sending IOP looks up the Target Address to find the alias TID the originating IOP assigned to the replying device and uses it to replace the Target Address.
11. The sending IOP allocates a message frame by reading the originating IOP's inbound message port. It places the message in the message frame and notifies the originating IOP by writing the address of the message frame to the IOP's inbound message port.
12. The originating IOP inspects the reply header's Initiator Context field and posts the message to the appropriate DDM.
13. Later, the DDM releases the message frame and the IOP replaces its address on the free list.

2.1.6.4 Flow of Events

Figure 2-14 illustrates the flow of I/O operations. The text following the figure describes the events (its numbers correspond to the steps).

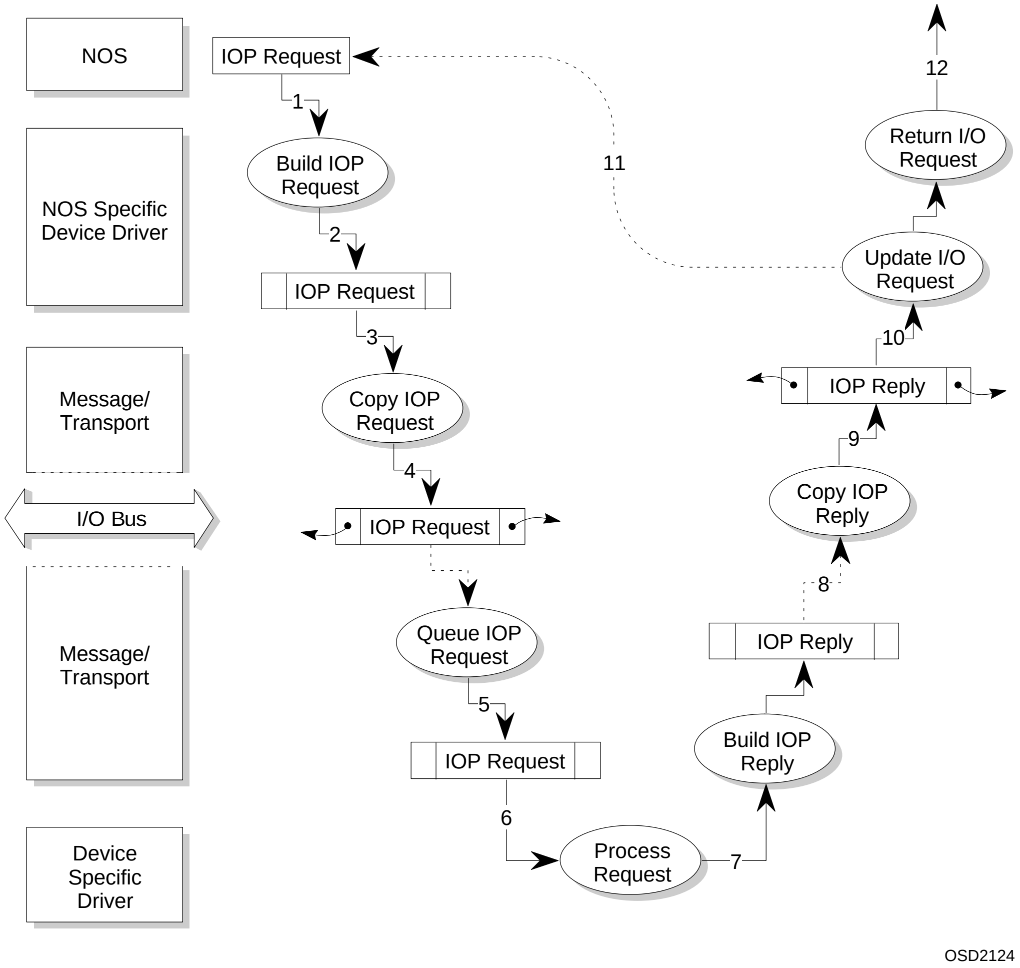


Figure 2-14. Flow of I/O Operations

1. The operating system issues an I/O request.
2. The OSM accepts the request and translates it into a message addressed to the DDM. The Initiator Context field is set to indicate the message handler for the reply. The OSM has the option to place a pointer to the OS I/O request in the message's transaction context field.
3. The OSM invokes the communication layer to deliver the message.
4. The host's MessengerInstance queues the message by copying it into a message frame buffer residing on the remote IOP.
5. The IOP on the other end posts the message to the DDM's event queue.
6. The DDM processes the request.
7. After processing the message and satisfying the request, the DDM builds a reply, copies the initiator's context and transaction context fields from the request to the reply, addresses the reply to the initiator, and finally invokes the message service to send it to the originator of the request.

8. The IOP's message service queues the reply by copying it into a message frame buffer residing at the host's MessengerInstance.
9. The IOP alerts the host's MessengerInstance to the message ready for delivery.
10. The host's MessengerInstance invokes the OSM's message handler with the reply.
11. The OSM retrieves the pointer to the OS I/O request from the message's transaction context field to establish the original request context and completes the OS I/O request.
12. The driver returns the request to the OS.

2.1.7 Configuring an I_2O System

The I_2O system contains the host and a number of IOPs. Each IOP is actually a subsystem composed of a MessengerInstance and a number of adapters managed by DDMs executing on the IOP. This section is concerned with configuring the IOP, installing DDMs onto that platform, and configuring those drivers. Furthermore, this section describes initializing the IOP, HDMs, and ISMs. Figure 2-15 depicts various IOPs' architectures.

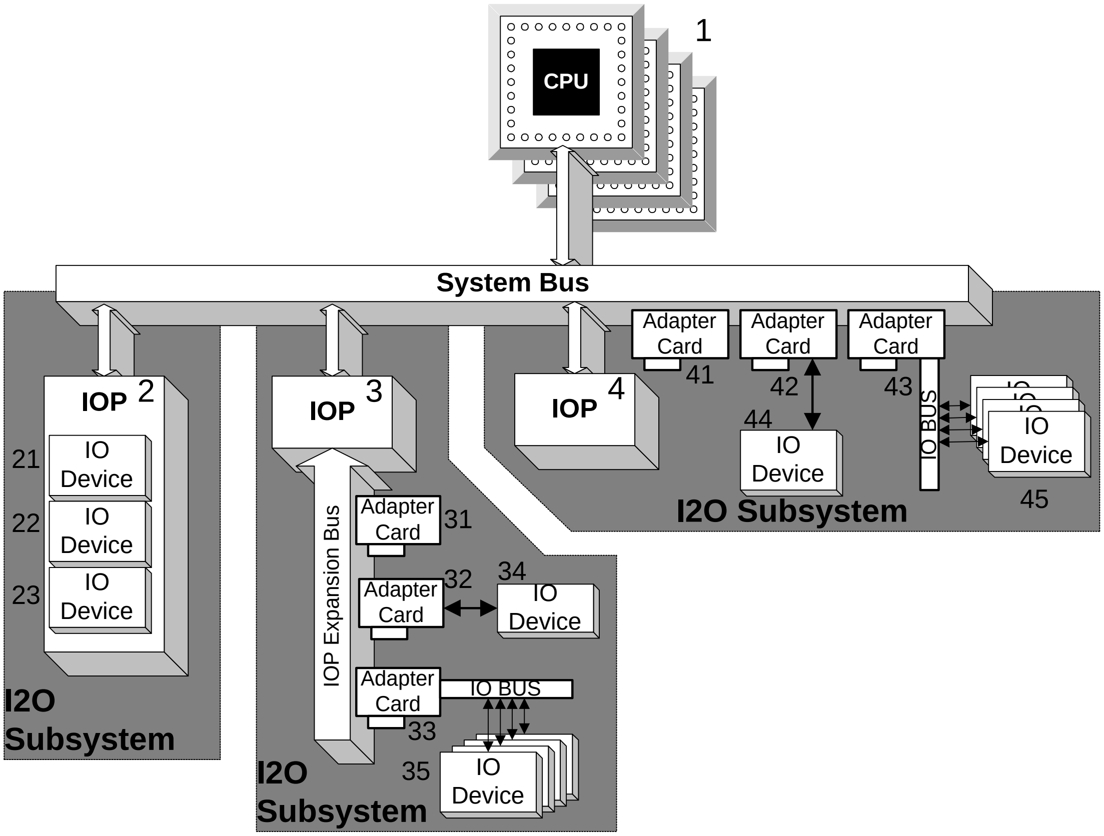


Figure 2-15. Various I_2O Subsystems

2.1.7.1 Architecture Variations

A goal of the I_2O specification is allowing flexible and extensible implementations. Many architectural variations are supported, with the following considerations:

- An IOP is an I/O processing platform that can be integrated into the system or an add-in card.
- The IOP can contain embedded I/O devices (like IOP-2 in Figure 2-15) or control adapters provided by third-party vendors (feature cards).
- These third-party feature cards can be attached to the system I/O bus (like IOP-4) or an expansion bus provided by the IOP (like IOP-3).
- An adapter can contain the I/O device, provide an interface for a specific device, or provide yet another I/O bus (e.g., SCSI) that the user can populate with varying devices from other vendors.
- An IOP can contain any number and variety of adapters, devices, and expansion buses.

Providing for flexibility and extensibility, this section defines the functions and interfaces necessary to:

- install I_2O DDMs for third-party adapters
- configure the IOP
- configure the DDMs
- upgrade the IOP environment
- upgrade the DDMs

This specification also defines extensions for common I/O buses, such as PCI and SCSI, so a developer can design a DDM for an adapter or device on one of those buses that functions on any IOP that accommodates it.

2.1.7.2 IOP Configuration

An IOP has the following configuration characteristics and capabilities:

- self-booting, does not require user interaction
- self-initializing
- enables configuring the IOP
- provides storage for DDMs and their configuration information
- provides the configuration interface for its DDMs
- provides a logical configuration table that presents available resources to the host and other IOPs
- gives the host a hardware resource table containing the hardware configurations and capabilities of the IOP's adapters.

When an IOP is powered on, it loads and initializes its operating environment. The mechanisms used to load and boot the I_2O environment can depend on, and be specified by, extensions to the system BIOS. When an IOP is integrated with the system, the system vendor can use existing system resources and capabilities. When an IOP is an expansion card intended for deployment in existing systems, it must not expect the system BIOS to be I_2O aware. It must not expect any special system utilities beyond those provided for any other I/O adapter that resides in the same system.

In either case, a host driver in the form of a BIOS extension or an OS driver is required for the host to take advantage of the I_2O system. Again, loading the host driver is system-dependent and outside the scope of this document.

After the IOP loads, it executes its initialization sequence. Part of that sequence causes the IOP to scan its physical adapters, load and initialize appropriate DDMs, and build a logical configuration table. When the IOP scans its adapters, the IOP (or appropriate HDM) must detect any change in configuration and, for each change, request a configuration dialogue notifying the user of the change. During the configuration, the IOP interacts with the system, and ultimately the user, to establish configuration parameters, install new drivers, or upgrade current drivers. A DDM can also request a dialogue to establish configuration parameters for devices it controls. The dialogue takes place between the operating system and the DDM or IOP being configured. The operating system is responsible for invoking the configuration dialogue and can do so at any time, independent of whether the IOP or DDM requests it. This lets users reconfigure the I_2O environment whenever they want.

Each module’s configuration dialogue establishes features and capabilities unique to the vendor. For the IOP, this mechanism also assigns adapters and devices, and manages subsystem topology. The configuration dialogue is generally invoked during installation and is not required during normal system booting .

To help manage IOP drivers, each DDM is identified by an organization ID[1] assigned to the vendor, a vendor-assigned module ID, and a version number. The IOP maintains a list of physical adapters, their locations and associated DDMs. As the IOP locates an adapter, it loads and initializes the associated DDM into the operating environment, assuming it is not already loaded. The IOP then instructs the DDM to initialize the hardware. During the adapter initialization process, the DDM identifies ports and functions in the adapter and registers them with the IOP as I_2O devices. The IOP uses this registration to update its logical configuration table and identify the TID assigned to the registered device.

When the IOP identifies a device registering as assigned to an ISM that is not loaded, the IOP loads and initializes the ISM. The IOP then sends the ISM a message identifying the device. During this process, like the HDM, the ISM creates and registers additional I_2O devices. Again, the IOP uses the registration to update its logical configuration table and identify the TIDs.

2.2 I_2O System Operation Overview

2.2.1 OS Centric View of System

This section to be added at a later date. It will discuss how the OS sees the IOPs and uses its abstracted services.

2.2.2 Peer-to-peer Capabilities

This specification describes functional interfaces based on the current PCI bus specification and does not attempt to modify or strengthen that specification. The PCI specification stipulates

[1] An organization ID may be obtained by contacting the I_2O SIG.

electrical characteristics and operating behavior of a PCI bus. It further describes how electrical PCI segments may be arranged in a hierarchy. In such a hierarchy, the electrical PCI buses are connected by PCI-to-PCI bridges to form a logical PCI bus, or *subsystem*. The PCI subsystem supports symmetrical access[2] between any two devices, regardless of their positions. This peer PCI operation is guaranteed only within the subsystem.

The PCI specification allows multiple PCI subsystems on a single host without peer-PCI operation. This is common among implementations. This document considers each PCI subsystem an I_2O segment, as described in section 2.1.5. It considers peer operations transactions between IOPs in the same I_2O segment.

The remainder of this section will be added later and explain extensions to peer-to-peer capability. This specification uses the term *peer operation* for message passing between IOPs, and *peer-to-peer* for transporting data between IOPs.

2.2.3 IOP Operational Overview

An IOP provides the shell and, optionally, the core interfaces, introduced in section 2.1.3.2.

2.2.3.1 Shell Interface

The interface between the host and an IOP is a message queue. The IOP provides an inbound message queue that the host or any IOP can use to send messages to the IOP. The inbound queue is actually a pair of queues: a free queue that contains empty message frames, and a work queue where message frames are deposited for processing. The sender removes a message frame from the free queue, places a message in it, then posts the message to the work queue. The IOP retrieves a request message from the work queue, and, based on the target address, places the message in the event queue for the appropriate DDM. When that module releases the message frame, it is returned to the free queue. Certain messages are addressed to the messenger itself, which are queued on the messenger's event queue and processed the same way any module processes messages. These messages regard path initialization, configuration table requests, connection requests, software installation, and configuration dialogue messages.

2.2.3.1.1 Configuration Table Requests

Any messenger can request an IOP's logical configuration table. The table tells the initiator which I/O services are available. The request can specify an immediate reply or defer it until the IOP's logical configuration table changes.

2.2.3.1.2 Connection Requests

A connection request connects a DDM on one IOP and a device registered on a different IOP. IOPs locally assign TIDs to DDMs and devices. These TIDs are unique only within an IOP, not across platforms. The connection request sets up an alias TID to identify the initiator when sending messages between the two IOPs. OSMs are preassigned an alias TID universally reserved

[2] Except for configuration cycles. Access to configuration registers is protected, and only propagated from the top down. Since I_2O is a memory interface, peer I_2O operation only affects memory access, not access to I/O or configuration registers.

for OSMs, so the host does not send connection setup requests. These requests can come from any IOP.

2.2.3.1.3 Software Download

This request gives the IOP the executable code (e.g., a DDM) to either store and/or load and execute in the IOP environment. If the module replaces an existing module, it is not loaded until the next time the IOP is initialized. When a module is first loaded, it is tagged as *experimental*. The experimental tag is removed by a positive action by the host or the operator. An experimental module is loaded only once. If the experimental tag is not removed before the IOP is initialized again, the experimental driver is discarded and its last version recovered. When the new module is accepted, the experimental flag changes to operational, and the old version of the driver is removed from the IOP store.

2.2.3.1.4 Software Upload

This request lets the IOP back up its operating environment of DDMs to the host facilities.

2.2.3.1.5 Configuration Dialogue Messages

Configuration dialogue messages let the user control installation and configuration. This document specifies a configuration language based on HTML that enables each I_2O component to define its own configuration dialogue, independent of the system’s physical resources (see the configuration interface in Chapter 3). Each module can display configuration text and prompt the user for input. The host invokes a configuration dialogue for the IOP, or it can be targeted at one of the service modules (DDMs). Only the host can invoke the configuration dialogue. Modules request a configuration dialogue by setting a flag in the logical configuration table.

2.2.3.2 Core Interface

This specification also provides an internal interface called the *core interface*. It produces a generic open platform supporting loadable modules (DDMs) where all DDM services are abstracted. The core interface specification allows third-party drivers to execute in a well-known environment. This interface consists of I_2O API functions providing embedded kernel, configuration, messaging, and transport services. These APIs give DDMs access to required OS functions without exposing the actual embedded OS to the DDM. The API establishes a *cocoon* that makes the module independent of its surrounding execution environment.

2.2.3.3 IOP Operations

This section describes the functions the IOP must provide.

2.2.3.3.1 IOP Configuration

The IOP’s operational parameters, such as the number of inbound message frames, are set through a configuration dialogue with the host (see the configuration interface in Chapter 3). DDMs are installed by executive messages from the host, as is assigning physical adapters and connections between DDMs. The IOP contains a storage facility that saves the installed DDMs and remembers the configuration information between power cycles.

Other parameters are established during the IOP's initialization sequence. Typically, there are two initialization sequences: one when the BIOS brings an IOP on line to boot the OS, and a second when the OS initializes. If the IOP does not provide access to the boot device, the BIOS initialization might be bypassed. In any case, an IOP with hidden adapters may require the host to reserve additional system memory space and system I/O space where the IOP can configure such adapters. These operational parameters are established via the initialization messages.

2.2.3.3.2 IOP Environment Initialization

Initializing the IOP environment begins when the IOP loads its operating environment and performs an initialization sequence with the host. This initialization sequence guarantees that all system adapters, regardless of whether they are controlled by an IOP, are configured into the system memory and I/O space. It also provides the synchronization points to guarantee that each adapter is controlled by only one source.

For adapters that an IOP controls, it loads DDMs from its store, initializing and invoking those modules to initialize their hardware. As part of the adapter initialization, each DDM registers devices with the IOP, which uses this information to construct a configuration database that describes the logical configuration of the IOP. The information used to build the database comes either from configuration data structures already built by the IOP software (e.g., configuration files) or a physical scan of the hardware environment (e.g., PCI bus scan). The IOP uses its own physical configuration table to assure consistent and deterministic assignment of adapters to HDMs and assign TIDs to registered devices. The physical configuration table allows the IOP to detect when adapters and/or facilities are added, removed, or changed. Once the logical configuration database is functional, the I_2O environment can accept connections and service requests.

The bootstrapping process can involve all, some, or none of the following stages, depending on the type of IOP and its state when the system BIOS queries it:

- When the system BIOS is I_2O aware, it initializes IOPs as necessary to boot the operating system.
- Each IOP provides its own BIOS extension if it is to participate in the system boot process when the system BIOS is not I_2O aware. Each BIOS extension functions only with the IOP for which it was designed.
- If the BIOS cannot establish the IOP's state, resetting it brings it to a known state. The BIOS queries the IOP to locate a bootable device, either mass storage or a remote boot class device. The remote boot class is not defined in this version.
- For each block storage device that the BIOS attaches to the system, the BIOS updates the IOP's logical configuration table with the BIOS information (e.g., logical drive number) assigned to that device. The OS correlates I_2O devices with BIOS services using this information.
- If the boot device is a storage class device, the BIOS sends storage class messages to read the sectors from the disk. The BIOS updates the logical configuration table to indicate which TID it uses as the boot device.
- If the boot device is remote, then the BIOS sends RIPL class messages to load the executable boot file. The BIOS updates the logical configuration table to indicate which TID it uses as the boot device.

The boot process continues normally loading the host OS, which includes the OS' I_2O environment. The environment locates and initializes all IOPs and notifies them of the others in the system. As the host OS boots, the information in the IOP's logical configuration table provides the information the OS needs to continue booting. As with any device, switching control from the BIOS to the OS must occur in an orderly fashion.

2.2.3.3.3 DDM Initialization

A functional embedded I_2O environment must be established before any DDM can be loaded. Loading a DDM is initiated either locally by the embedded I_2O environment, as a result of locating a driver in its local store, or, remotely, in response to a download request.

During initialization, the DDM must update the IOP's logical configuration database, reflecting the actual devices found on-line and their current configuration. The DDM creates and manages I_2O devices. The I_2O specification supports dynamically loaded and unloaded devices, drivers, and I/O services in general. Maintaining this dynamic environment and tracking dependencies is the responsibility of the IOP's configuration manager.

2.2.3.3.4 Loading DDMs

The IOP supports loading I/O driver modules onto the IOP and initializing the module. Once the code loads, the IOP invokes the module's initialization code (MAIN). This initialization binds the module to the IOP, providing the API entry points and enabling the module to call the IOP's service routines. Each module registers with the IOP, which assigns a TID.

2.2.3.3.5 DDM Module Registration

During its registration, a module describes for its attributes and registers a set of entry points known as *message handlers*. It calls a registration function of the local IOP, which places that information in the IOP's configuration tables as a special DDM class device. The IOP issues a TID to the DDM. The IOP dispatches the appropriate message handler for processing the message when a request addressed to that TID is received. The message handlers registered by the module are I/O class neutral; DDM class messages are generic and independent of the actual I/O class of the DDM. This means that a LAN DDM and a SCSI DDM both register a DDM class TID, and messages sent to a DDM class TID are neither LAN- nor SCSI-specific. A DDM registers additional TIDs as class-specific devices.

2.2.3.3.6 HDM Adapter Initialization

A DDM can control multiple adapters, and the IOP must configure each DDM specifically for the adapters it controls by binding the module to the hardware. The IOP issues the DDM a message for each adapter the DDM will control. As the DDM brings an adapter or service online, it registers that service and any I/O devices with the IOP by registering (creating) I_2O devices. This routine repeats for each adapter the DDM will control.

2.2.3.3.7 ISM Device Initialization

A DDM can control I_2O devices as well as adapters, and each DDM must be specifically configured for the devices it controls. This configuration occurs during the IOP's initialization sequence. Where an adapter can only have one DDM controlling it, an I_2O device may have many clients (service users). The primary client determines whether the service may be shared with other clients.

For each device registered, the IOP issues an attach message to each DDM that identifies that it can control that class of device. That DDM then sends the device a claim message to acquire its service. The IOP must attach devices to DDMs consistently, so that the primary user can acquire the device first. As the DDM acquires devices, it may register additional devices with the IOP as appropriate. This routine repeats for all devices registered.

2.2.3.3.8 I/O Device Registration

As a module initializes an adapter, it creates an I_2O device for each function or service provided by that adapter. This registers the service, placing the registration information in the IOP's logical configuration table. Some information disclosed during registration describes the class of service. Each device is registered with a specific class. The class of service is the primary search criterion for any user looking for a service provider. The registration message also establishes the message handlers for that particular device instance. The IOP replies to the registration, assigning or verifying the TID for that device. This registered service is now an I_2O device that can be addressed through its assigned TID. The TID directs messages to the DDM's handlers managing the device. The TID registered for the device is class specific; requests to a TID imply a particular class.

2.2.3.3.9 Messenger Service

Connection management. The host may assign devices registered on one IOP to an ISM on another. Before an IOP can attach a remote device to a DDM, the local IOP sends a connection setup request to the remote IOP, establishing the connection and assigning alias TIDs. Once the alias TIDs are assigned, the IOP can attach the remote device to the DDM. The DDM does not know the device is remote.

Messenger service. When a message arrives in the IOP's inbound queue:

1. The IOP dispatches a request based on the target address in the message header. The message is received with the remote DDM's locally assigned alias TID (assigned by this IOP to the remote DDM) in the initiator's address field. The TID locally assigned to the target module is in the target address field.
2. The IOP dispatches replies based on the Initiator Context in the message header.

The IOP places the message in the target's event queue and schedules the target's message handler. After the target releases the message, the IOP places the empty message frame on the free list to be used again.

When a DDM sends a request to another IOP, the local IOP replaces the alias target address with the actual TID assigned by the remote IOP. It also replaces the initiator address with the alias TID assigned by the remote IOP for that initiator. The IOP then places the message in the target IOP's inbound queue.

When a DDM replies to another IOP, the local IOP replaces the alias initiator address with the actual TID assigned by the remote IOP. It also replaces the target address with the alias TID assigned by the remote IOP for that initiator. The IOP then places the message in the target IOP's inbound queue.

If the DDM targets a message for a local module, the IOP posts the message to the target's event queue without modifying the address fields, and schedules the target's message handler. If the initiator address is the TID reserved for OSMs, then the IOP posts the message to the outbound message queue without modifying the address fields.

2.2.4 DDM Operational Overview

A DDM consists of several functions and represents multiple I_2O devices. The top component of a DDM is its class device, which the DDM creates when the IOP initializes the module. When the DDM creates its class device, it specifies message handlers that the IOP calls to process messages to the DDM. These messages are DDM class messages (see Chapter 5).

The IOP assigns (attaches) and identifies an adapter to the DDM by sending it a message. This message authorizes the DDM to access and initialize the adapter. For each instance of service the adapter provides, the DDM creates an I_2O device. As an example, the IOP attaches an adapter card containing two SCSI ports to the vendor's DDM. The DDM creates two additional devices (bus adapter class) for the ports, and each has an assigned TID. When the DDM creates the bus adapter device, the DDM specifies message handlers that the IOP calls to process messages to that TID. These messages are bus adapter class messages (see Chapter 6). The DDM also scans for SCSI devices attached to each SCSI bus, and for each device found, creates a SCSI Peripheral class device.

The DDM also creates objects to service each interrupt. The DDM provides a routine that services the hardware and clears the interrupt. It executes in the context of the interrupt thread. This routine may also post an event that executes in the DDM's context.

The model for a DDM is a number of event handlers designed to be non-blocking. That is, the event handler executes continuously and quickly until it completes all available processing. During the execution, the DDM does not wait for hardware or facilities to become available. Thus, a routine is split into a number of event handlers, so that when the routine reaches the point where it would normally wait, it instead defers. When the hardware or facility is available, the next event handler is scheduled to continue processing the task.

2.2.4.1 Module Structure

DDM module structure is described in detail in Chapter 5. The DDM is a set of event handlers. Each device is associated with an event queue. When an event occurs, such as a message arriving, the IOP creates an event and posts it to the event queue. When the event reaches the head of the queue, its event handler is invoked. In general, for each I_2O device that the DDM registers, it provides message handlers that process the messages to the device. The DDM also provides an interrupt handler for each discrete interrupt. If the DDM calls an IOP service that must defer until an external event or another operation completes, the DDM supplies the identity of the event handler that continues the processing once the event occurs. This is generally accompanied by a context variable so the handler can determine which transaction spawned the operation. That way, the DDM processes multiple functions in parallel.

2.2.4.2 IRTOS Services

The IOP supplies services that support the DDM's operation. The IOP abstracts these services, called IRTOS services, as a set of APIs. These services include memory allocation, DDM and device registration, DMA operation, bus access, and timer operation.

2.2.4.3 Messenger Service

The IRTOS supports dispatching received messages as well as delivering messages the DDM creates. For each device registered, the DDM gives the IRTOS a list of message handlers, one for each message function, and their respective priorities. The IRTOS builds a message dispatch table so that when a request arrives, the IRTOS looks up the message function and posts an event for the corresponding event handler at the appropriate priority.

2.2.4.4 Transport Service

The IOP abstracts the hardware to the DDM, so the DDM must call API functions to access hardware. The IRTOS provides API functions for accessing configuration registers, I/O ports, and memory of a particular adapter or system memory. The IOP also provides DMA objects that more efficiently transfer blocks of data.

2.3 Message Service

The communication model used by the I_2O architecture is a message-passing protocol. The communication service, as defined in this document, provides the message transport service for OSMs and DDMs. This message transport service abstracts the I/O infrastructure, leaving the service interface to those modules independent of the system hardware.

2.3.1 Conceptual Overview

The I_2O communication layer enables modules to exchange messages without knowing the underlying bus architecture or system topology, as illustrated in Figure 2-3. An I_2O domain consists of any number of MessengerInstances. An IOP can support up to 4087 registered I_2O devices (I/O driver modules and registered services). Each IOP contains an I_2O resource manager that facilitates the interface between IOPs, directs configuring and initializing DDMs, and supervises the connections between them.

Together, the MessengerInstances form a network that can deliver I/O transaction messages from one module to another, anywhere in the I_2O domain.

2.3.2 Communication Model

The messages themselves are referred to as *message frames*. Each message frame contains a *message header* and a *message payload*. Message header format is defined by the I_2O messaging layer. The message header format is constant for all messages and provides the return address of the originator. The format for the message payload varies between messages and is established by the function type value in the header. Each class defines its own message payload formats.

In addition, this specification defines a neutral memory buffer descriptor format (i.e., a scatter-gather list) that provides independence from the host operating system memory model.

Each device is a virtual interface for a particular class of I/O messages. A TID identifies a device and, thus, an instance of a class-specific interface. Only requests of that class may be sent to that TID. The IOP administers TIDs when a device is first created, and the TID acts as the local address of the device.

2.3.3 Communication Architecture

All communication is performed through a TID. A message carries the addresses of the initiator and the target. Messages -- either requests or responses -- travel from one service module to another. Responses are addressed to the initiator of the request. Specifically, an OSM or DDM sends a request to a device, which replies to the initiating module. The class of message is determined by the device's registered class.

2.3.3.1 Memory Environment

All memory structures (i.e., message frames) shared between MessengerInstances must reside in physically contiguous memory. They can lie in different memory locations among the processors, but the memory of each structure must be physically contiguous. This means the memory of each target message frame -- where the message resides so that the handler can process it -- must be contiguous.

Data referenced in message frames must reside in memory shared by the target and the initiator. If the initiator and target are on different IOPs, memory must be referenced by its system memory reference.

2.3.3.2 Peer Communication

The ability to exchange alias TIDs is the only additional function necessary to send messages between IOPs. Each IOP is a master in terms of dispatching messages. Each initiator has equal priority; this enables peer communications.

2.3.3.3 Protection and Marshaling

The messaging layer does not enforce any rules structure or content over that of inspecting the initiator and target addresses for routing. The communication layer expects that each module sends the proper class of message on each path.

2.3.3.4 Message Header

Think of the message header as the envelope of a mailed letter. It provides just enough information to route the message and dispatch it to the appropriate module.

2.3.3.5 Target IDs

TIDs are assigned by each IOP. TIDs must be unique within an IOP. An IOP must assign TIDs consistently so that, without physical change, the TIDs assigned to local devices are the same each time the IOP is initialized. For example, a disk drive maintains the same TID in the IOP's logical configuration table. Once the IOP assigns a TID, it should not reassign it to another device.

2.3.3.6 Endian Support

This version of the specification discusses operation for little endian addressing only. Big endian addressing may be specified in a future revision. The Message Version field supports future capabilities, such as big endian messages.

2.3.3.7 64-Bit Addressing

Three domains affect address size: the OS, the I/O subsystem, and the IOP. This version of the document specifies operation for 32-bit IOP physical addressing, 32-bit I/O subsystem operation, and both 32-bit and 64-bit OS operation. The OS address size relates to the size of the message context fields. Future versions of this specification may specify 64-bit physical addressing for 64-bit I/O subsystems. This version adds hooks (to the SGL) to support 64-bit physical addressing. The Message Version field in the message header supports future capabilities, such as 64-bit physical addressing.

Critical messages for initializing the IOP are address-size generic, allowing the OS to appropriately instate the IOP into the system.

2.3.4 Dependencies

- IOPs must have access to shared system memory for the queuing model described.
- Each IOP must provide its own units for receiving messages from the host and other IOPs and for queuing messages to the host.
- Efficient memory coherency support is required if shared memory writes involve a write-to cache, versus a write-through or copy-back. If so, an efficient mechanism to flush modified cache lines must be provided.

2.4 Configuration Dialogue

The configuration dialogue allows configuring hardware-specific parameters, as well as enabling and disabling vendor-unique features. The configuration language, based on HTML, lets each vendor design its own user display using a standard language. The IRTOS provides several API functions that aid the DDM's dialogue and also accommodate internationalization. Internationalization will be fully defined once HTML supports it.

The configuration dialogue is always invoked by the host and can be requested by any DDM or IOP. The requester sets a flag in the IOP's logical configuration table.

2.5 Profile of an Intelligent I/O Platform

This section demonstrates an IOP built to I_2O requirements. The discussion also includes the facilities expected by a DDM and their use. This section provides models to help develop the hardware platform, the IRTOS, and the DDMs. These models are only a guide and should be construed as neither an architectural requirement, nor the only models that satisfy the I_2O architecture.

Even though this specification attempts to maintain hardware independence, this discussion focuses on PCI for the following reasons:

- It is difficult to describe abstract requirements and still articulate value or convey opportunity.
- PCI is currently a pervasive I/O bus technology.
- Optimum efficiency demands more stringent technical requirements. This is the case with PCI. No other technologies are identified, so assume that some optimizations may not apply when such technology is identified.
- An IOP may incorporate other expansion bus technologies, but optimization depends on the bus characteristics. The effort accommodates adapters and their DDMs from third-party vendors. These optimizations will be addressed as other interesting and valuable technologies develop.

2.5.1 Data Movement

Understanding data movement is crucial to understanding the IOP's hardware requirements and I_2O system operation.

2.5.1.1 Concepts of Data Movement

The IOP's principal task is moving data between either main system memory and one or more devices, or between multiple devices. The IOP achieves this functionality through combined hardware and software. To realize the required functionality, the IOP must meet certain minimum hardware requirements. An IOP implementation involves many adapters on multiple buses. Accordingly, an important function of hardware is to move data from one point to another. This, in turn, affects the behavior of hardware associated with the various buses.

Moving data through a system implementing the I_2O specification occurs through:

1. asserting addresses and controls on a bus
2. claiming the address by some element on that bus
3. retrieving or asserting data on the bus according to the control

These operations may be cascaded or nested. That is, the claimant on one bus may initiate a transaction on another bus to dispose of or obtain data. At least two forms of cascading can be identified. First, some elements may, in effect, serve as repeaters. (Bridges, at least the simplest ones, fall into this category.) These elements claim an address on one bus and reassert it (unchanged) on another. Second, elements may claim an address on one bus and reassert it in some modified form on another. The first case involves multiple physical buses, but only one logical bus. All physical buses connected to it are considered as having a common address space. In the latter case, all distinct buses, both physical and logical, have their own address spaces.

2.5.1.2 Components of Data Movement

In an I_2O system, three classes of buses are distinguished:

1. the system bus

3. an IOP local bus for each IOP

4. zero or more expansion buses per IOP.

Hardware moves data by referencing its address. Each bus has a unique address space. Thus, an I_2O system contains three noteworthy classes of address space.

1. Because there is only one system bus, there is likewise only one *system address space*.
2. A given IOP typically has only one local bus. Therefore, each IOP has a *local address space*.
3. Finally, a given IOP may support multiple expansion buses. Therefore, a given IOP may have several *expansion bus address spaces.* Across the system there may lie many expansion bus address spaces.

Several types of elements are involved in executing intelligent I/O. These elements, distinguished by how they interact with buses, include:

- ***Processor elements***. Processors assert addresses, and either retrieve or assert data on the bus to which they attach.
- ***Memory elements***. Memories claim addresses within a range associated with the element and either pull data from the bus for storage, or retrieve data from storage asserting it on the bus.
- ***Access unit elements***. Access unit elements claim addresses in a range associated with the element, translate them according to a rule (i.e., linear address mapping). They reassert the addresses and corresponding controls on another bus. Also, by implication, the access unit either retrieves data from the source bus and reasserts it on the target bus, or vice versa. Also, by implication, access units are unidirectional. Bridges, in essence, are a pair of access units operating in opposite directions. The address ranges of each unit are mutually exclusive.
- ***DMA engines***. DMA engines affect moving data from a source on one bus to a destination on the same or another bus. A DMA engine is programmed with a source address, a destination address, and a data length (or the logical equivalent). The DMA engine then affects data movement in some increment from the source to the destination, incrementing the source and destination addresses accordingly, until the prescribed amount of data is transferred.

  If a DMA engine spans buses, the source address is in the address space of the source bus. The destination address is in the address space of the target bus. On both the source and target buses, the DMA engine asserts an address and control. The claimants of the source/target addresses should deliver/retrieve the corresponding data accordingly. Logically, a DMA engine does not actually touch the data, although hardware optimizations may provide intermediate buffering, and so forth.
- ***Bus master elements***. Bus master elements (like processors) assert addresses and data and/or retrieve data accordingly. Addresses asserted by bus masters are in the bus where the adapter resides, and are implicitly claimed by some other element on the bus (memory, access unit, or a bus slave). Most bus master elements are also bus slave elements.
- ***Bus slave elements***. Bus slave elements claim addresses (like memory elements) and assert or retrieve data accordingly. Addresses claimed by a slave element are configured to the element and are in the address space of the bus where the element physically resides. Adapter control and programmed I/O operations to I/O adapters are examples of bus slave elements. Other elements may also contain a slave interface (e.g., access units to set up translation rules, DMA engines and bus master adapters to set up transfer parameters).

On any given bus, addresses can be classified as *local* or *remote*. A local address is asserted on the bus claimed by another element on the same bus that is the ultimate destination. A remote address is claimed by an access unit and, based on rules programmed into the access unit, is delivered to another bus. Having no claimant is possible, but usually invalid. Any address on the originating bus should have only one terminal claimant throughout the address spaces in the system. Ultimately, a terminal claimant either consumes or provides data (e.g., a memory element). Thus, across the entire I_2O system, address mappings must ensure only appropriate mappings among address spaces.

When asserting and claiming addresses:

- Processors reside on a specific bus and can assert any address that bus allows. A host processor resides on the system bus and an I/O processor resides on the IOP's local bus.
- Bus master adapters reside on a specific bus and can assert any address that bus allows.
- Memories reside on a specific bus and claim addresses within ranges designated for that purpose. Memory on the system bus is referred to as main system memory.
- Slave adapters reside on a specific bus. Slave adapters claim addresses within ranges designated on the bus.
- Access units reside on two buses, with a source connection on one bus and a target connection on another. The source connection claims addresses within ranges designated on the source bus. The target connection can assert any address the target bus allows.

2.5.1.3 I_2O Data Movement Components

The I_2O specification implies minimum hardware requirements for moving data throughout the system. Every IOP must implement the following functionality:

- ***System/local access unit***. An access unit with the system bus as its source and the IOP's local bus as its target. This implies that some range of system address space is mapped onto a region of each IOP's local bus address space. Claimants for such system addresses are either the inbound and outbound message queues, or the IOP local memory (e.g., inbound message frames). The mapping mechanism for IOP memory is a simple offset. That is, the linear mapping from system to IOP local bus address requires a (possibly negative) constant in the system bus address.
- ***System DMA unit***. A logical device connecting the IOP's local and system buses, which transfers data between any two addresses that can be claimed on the local and system buses. If an IOP implements the I_2O runtime environment for third-party DDMs, the hardware must support logical DMA functionality defined by the IRTOS API specifications. This may be achieved through a suitable hardware DMA engine or however allows proper operation of the relevant API.
- ***Local/system access unit*** (conditional). If an IOP implements the I_2O runtime environment and allows connecting third-party bus master adapters to the IOP's local bus, it must also provide an access unit whose source is the IOP local bus and whose target is the system bus. Bus master adapters on the IOP local bus must have direct access to system memory. Mapping from a local to a system bus address requires a (possibly negative) constant in the IOP local bus address.
- ***Expansion bus DMA/access unit*** (conditional). If an IOP implements the I_2O runtime environment and supports connecting third-party adapters to one or more expansion buses, it must also provide a logical DMA element connecting the expansion bus and the IOP's local bus. That element transfers data between any two addresses that can be claimed on the expansion bus and the IOP's local bus. IOP hardware must support logical DMA functionality

defined by the IRTOS API specifications. This may be achieved using either a suitable hardware DMA engine or another way that allows operation of the relevant API.

In addition, a set of single access API transport functions enables the IOP to provide a mechanism whose target is the expansion bus. It transfers data to or from any address that can be claimed on that expansion bus. IOP hardware must support the logical transport functionality defined by the IRTOS API specifications. This may be achieved either using a suitable hardware DMA engine or in another way that allows operation of the relevant API. Unlike the expansion bus DMA, this function is optimized for small transfers, such as control data and commands. It may be the same facility operating at a higher priority. Note that DMA API functions queue the DMA task and return, in contrast to transport API functions that are blocking.

- ***Expansion bus/system access unit*** (conditional). An IOP may implement the I_2O runtime environment and allow connecting third party bus master adapters to one or more expansion buses. If so, an access unit, whose source is the expansion bus and whose target is the system bus, is required for each expansion bus. That enables bus master adapters on the expansion bus to access system memory directly. The linear mapping from the expansion to system bus address requires adding a (possibly negative) constant to the expansion bus address. For PCI this constant must be zero (0).
- ***Expansion bus/local access unit*** (conditional). An IOP may implement the I_2O runtime environment and support connecting third party bus master adapters to one or more expansion buses. If so, each expansion bus requires an access unit with the expansion bus as its source and the IOP’s local bus as its target. This allows the bus master adapter on the expansion bus to access a portion of the IOP’s memory directly. The linear mapping from expansion bus address to the IOP’s local bus address requires adding a (possibly negative) constant to the expansion bus address.

2.5.2 Fundamental Elements

The simplest model for an IOP is shown Figure 2-16. This model fits an intelligent I/O feature card with its own embedded I/O ports or devices. Applications might include an intelligent Ethernet card or a RAID adapter.

In this model, the fundamental elements (non-shaded, heavy border) support embedded slave I/O devices. The IOP’s local memory is logically partitioned into three regions defined by their access capability. From a DDM’s viewpoint:

1. The Code Partition represents read-only memory.
2. The Private Data Partition is accessible only via the processor data commands.
3. The Shared Partition is also accessible via the host and other IOPs (the effect of the system/local access unit). Inbound message frames reside in the Shared Partition.

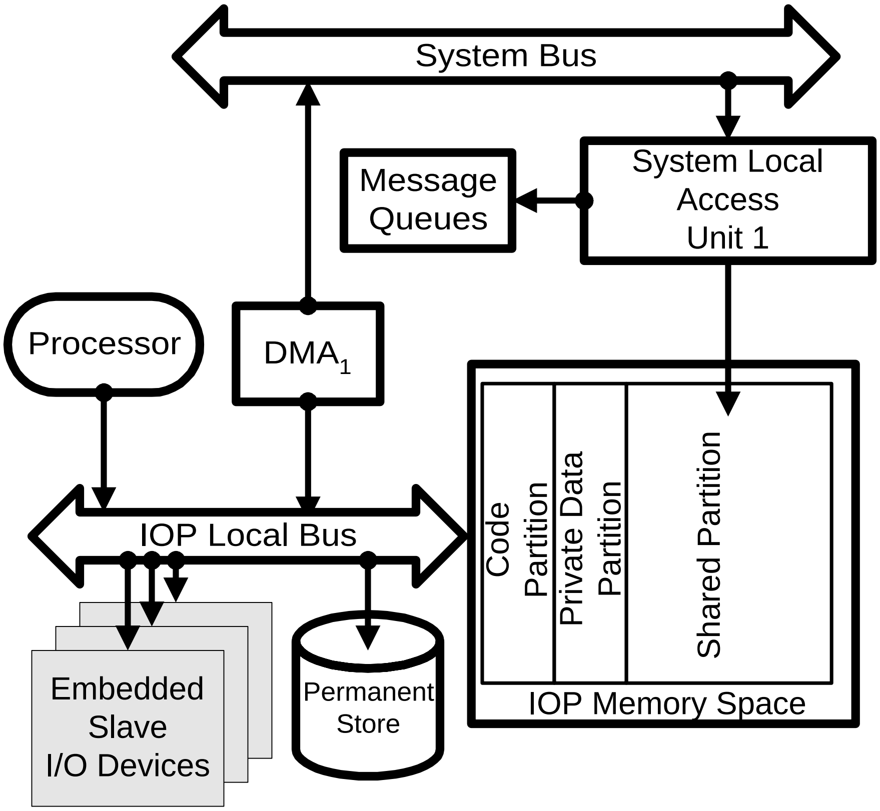


Figure 2-16. Simple Form of an Intelligent I/O Platform

In this model, the I/O devices reside directly on the local bus. The only additional facility necessary for data transfer is DMA$_1$. Using DMA$_1$, a DDM (via IRTOS DMA function calls) transfers data directly from a device or local memory to system memory specified by a request. The IRTOS also utilizes DMA$_1$ when it moves an outbound message to an outbound message frame located in the system memory or to another IOP's inbound message frame.

Shaded elements indicate components that add value beyond the minimum requirements of this specification. A brief description of each component and its required functions and capabilities follows.

2.5.2.1 Processor

The processor is the key to performance and capacity. This specification imposes no requirements on the processor other than the scope of its tools. It strives to make DDMs written in C code portable across platforms. Therefore, if the platform supports loadable DDMs, the processor's tools must include a C compiler that is readily available to DDM writers. The processor, with the other IOP components, must provide the API functionality specified in Chapter 5.

2.5.2.2 Permanent Store

The IOP provides non-volatile storage for its own code (i.e., bootstrap and IRTOS), drivers and embedded I/O functions, and for installed DDMs. For each DDM, the IOP stores the module's header, executable code, and a module parameter block. The IOP may use any form or technology to provide this capability. The amount of permanent storage and its implementation is left to your discretion for market differentiation.

2.5.2.3 IOP Local Bus

Two bus objects comprise the simple model: the system bus and the IOP's local bus. A DDM accesses memory and devices directly on the IOP's local bus, which is transparent to normal operation. However, a number of facilities provide access between the IOP's local bus and either the system or expansion bus, as illustrated in Figure 2-18. The IOP creates a bus object for its local bus. The private data region is specifically associated with this object, although all IOP local memory regions have an IOP local bus attribute. See section 2.5.4.3, *Bus Objects*.

2.5.2.4 System Bus

The IOP creates an object for the system bus. DMA$_1$ and a shared system memory region are specifically associated with this bus object. See section 2.5.4.3, *Bus Objects*.

2.5.2.5 System/Local Access Unit

A *system/local access unit* (ATU) claims memory transactions on one bus and translates them into memory transactions on another bus, substituting a different target address. The target is calculated by adding an offset to the original address. This allows the processor or bus master on the initiating bus to access a portion of the memory on another bus.

The ATU maps a portion of the IOP's memory into the system address space, creating shared memory. This unit allows the host (and other IOPs) to access inbound message frames as well as other shared structures. This facility also makes the message queues accessible. The configuration of this unit is specified by the system platform, rather than this specification. The ATU creates a window in the system memory space that accesses a block of IOP local memory. The IOP is expected to know the base address of that window, its size, and the local address for the block of local shared memory. This shared memory region has an attribute specifying its offset (i.e., the difference between the window base system address and the local address of the partition). See section 2.5.2.8, *IOP Local Memory Partitions*.

2.5.2.6 DMA

This function allows the processor to access system memory (including shared memory on other IOPs). The DMA is programmed by the processor to move blocks of data between system memory and the IOP's local memory. This is how the IOP fills outbound message frames (in system main memory) and inbound message frames of other IOPs. A DDM utilizes DMA units via DMA API function calls.

2.5.2.7 Message Queues

This function notifies the IRTOS when a message is deposited in its inbound queue. It is described in Chapter 4. A DDM does not directly access the message queues. The IRTOS receives messages and posts them to a DDM's event queue. The DDM uses API function calls to send messages.

2.5.2.8 IOP Local Memory Partitions

The IOP's local memory space is divided into partitions. Each partition represents a memory range with common access capabilities. Memory partitions can overlap and, in some cases, must do so. Each partition is associated with specific access capabilities. When requesting memory allocation, a DDM specifies a particular memory partition by specifying its access capabilities.

The following sections describe buses that also require access to IOP memory. The region of memory accessible from various buses is the *shared* partition. It is further subdivided based on which buses can access it. The region accessible from the system bus is the *shared system* region. The *all local adapter* region is accessible to any adapter controlled by the IOP. The *all adapter* region is where the shared system and all local adapter regions overlap. As opposed to the all local adapter and all adapter regions, the all adapter region must be accessible by adapters on other IOPs (if the IOP supports peer connections). However, if the IOP supports adapters on the system bus, the all adapters and all local adapters regions could conceivably be the same.

Memory partitions have the following attributes:

- Partition Size
- Local Bus Base Address
- System Bus Base Address
- Expansion Bus Base Address (for each expansion bus).

The IOP's processor (thus each DDM) accesses all partitions via the local bus, and therefore the DDM uses the local bus address to access that partition directly.

As illustrated in Figure 2-17, the all local adapter region is directly accessible to bus master adapters on an expansion bus via a bus/local ATU. In this case, the Expansion Bus Base Address indicates the offset of where that partition appears in that bus' address space (see section 2.5.3.6, *Expansion/Local Access Unit*).

- **Code partition**. The code partition(s) contain the executable image of the IRTOS and DDMs. Code partitions are generally protected and not available for access by DDMs or external agents. That is, DDMs can not allocate memory in a code partition.
- **Private data partition**. A portion of the IOP's memory that does not need external access. This partition is associated with the local bus object, because only the IOP's local bus needs access to the partition. The DDM specifies the BusID of the local bus when it allocates memory within the private data partition.
- **System shared partition**. The first level of shared memory is shared with the system (see section 2.5.2.5, *System/Local Access Unit*). This memory is characterized by its base address in the local address space (Local Bus Base Address) and its window position in the system memory space (System Bus Base Address). The system shared region is actually subdivided into two regions: One that can be accessed by a bus master adapter on a particular expansion bus through an inbound ATU (see section 2.5.4.2, *IOP Memory*) and one that cannot. The DDM allocates memory in the shared partition by specifying *System* access.
- **Adapter Shared partitions**. The next level of shared memory is accessible via adapters on the expansion buses (see section 2.5.3.6, *Expansion/Local Access Unit*). The all local adapter region may overlap with the Shared System partition, depending on the IOP's ability to control adapters on the system bus and devices on other IOPs.

2.5.3 Additional Elements

Add-in cards require an expansion bus. A model for an IOP that provides an expansion bus and, thus, supports third-party adapters and drivers is shown in Figure 2-17. A more complex model that provides multiple expansion buses is shown in Figure 2-18. Figure 2-18 shows three expansion buses and the additional resources needed to support them. Again, the fundamental elements have a bold outline and the additional elements are shaded. The three levels of shading indicate the elements associated with each expansion bus.

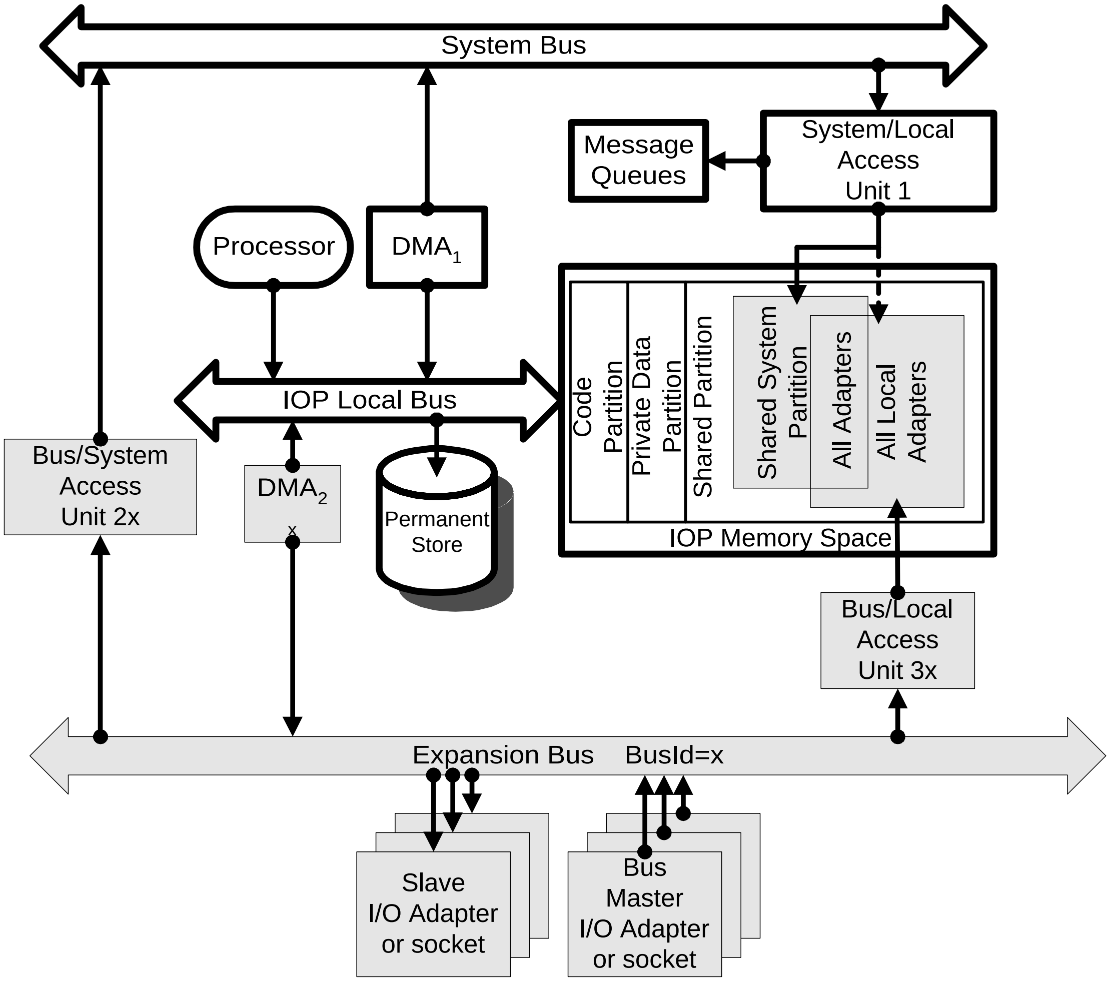


Figure 2-17. Typical Intelligent I/O Platform with Expansion Bus

To support slave I/O adapters, the DDM requires a DMA facility for transferring data between the adapter and local memory, or between the adapter and system memory. This facility is represented by DMAx in Figure 2-17 and DMAx, y, and z in Figure 2-18.

The bus/local access units (3x, 3y, and 3z) support bus master adapters on the expansion bus by providing local memory that the adapters can access. See section 2.5.3.6, *Expansion/Local Access Unit*. Each unit may have a different expansion bus base address, but must access the all local adapter region of memory.

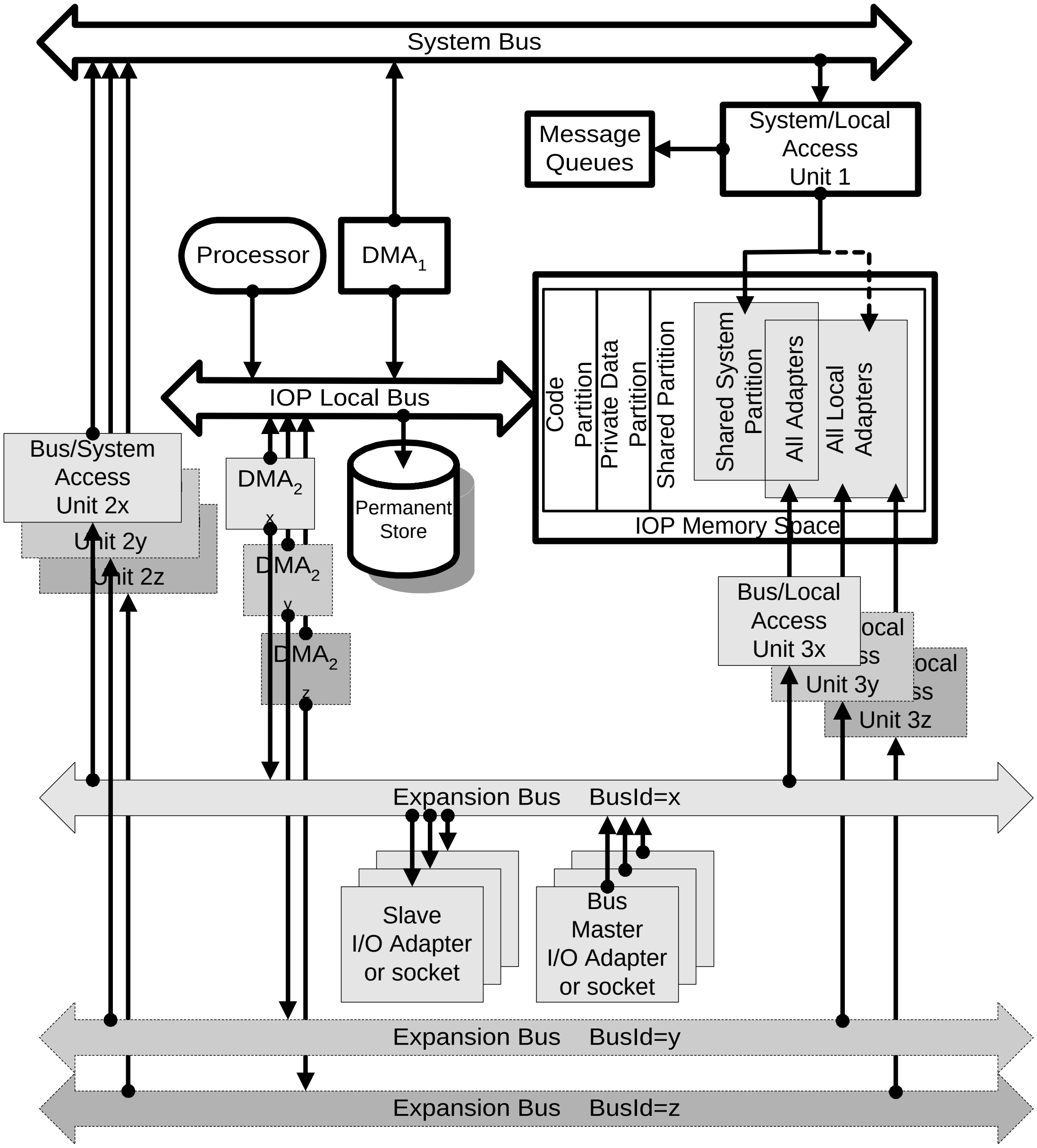


Figure 2-18. Complex Intelligent I/O Platform

2.5.3.1 Expansion Bus

The expansion bus is the key to an open platform. It must follow a standard bus architecture, such as PCI, that accepts adapter cards or controllers already designed for classic computer systems. Each expansion bus is identified by a BusID and has associated with it a number of facilities. To support slave adapters, the IOP provides a DMA mechanism, DMAx in Figure 2-17 and DMAs 2x, 2y, and 2z in Figure 2-18. See section 2.5.3.4 *Local DMA*. Bus master support requires two address translation units, as illustrated by bus/system access unit 2x and bus/local access unit 3x in Figure 2-17. (See sections 2.5.3.5, *Expansion Bus/System Access* and 2.5.3.6, *Expansion/Local Access Unit*.)

2.5.3.2 Slave I/O Adapter/Socket

A slave adapter is characterized as a block of memory and/or a set of I/O ports that the driver accesses to control the adapter and transfer data. The DDM accesses the device via API function calls (DMA, BusRead, or BusWrite).

The DMA capability quickly and efficiently transfers data blocks between either a slave adapter and local memory, or the adapter and system memory. The IOP's data transfer between the slave adapter and the system bus is abstracted to the driver. The IOP may have a direct engine or manage some intermediate buffers to transfer data from the expansion bus to a local buffer (or vice versa) and system memory. The attributes of a DMA object include the BusIDs of the source and target buses.

2.5.3.3 Bus Master Adapter or Socket

A bus master adapter adds to the complexity. It must still be programmed via the driver, generally by reading and writing to memory-mapped I/O registers or ports. Therefore, the same facilities that access the slave adapter are also necessary for a bus master adapter. In addition, the bus master adapter must be able to transfer data directly into buffers in main system memory. This requires a data path provided by the bus/system access unit 2x, or an address translation unit. See section 2.5.3.5, *Expansion Bus/System Access*.

2.5.3.4 Local DMA

Local DMA must be able to transfer large blocks of data efficiently between the IOP's local memory and an adapter on an external bus, illustrated by DMA2x in Figure 2-17, and DMA2x, 2y, and 2z in Figure 2-18. An IOP needs DMA mechanisms that can serve all expansion buses and the system bus. The IOP may have a direct engine that transfers data between any two buses, or move the data from one bus to a local buffer, and then to the target bus.

2.5.3.5 Expansion Bus/System Access Unit

As mentioned previously, bus master adapters must transfer data directly in and out of system memory. This is illustrated by Unit 2x in Figure 2-17 and 2x, 2y, and 2z in Figure 2-18. This address translation unit claims addresses on the expansion bus within a particular address range (or set of address ranges) and translates them to a memory transaction on the system bus. Addresses not claimed by an ATU are considered private address ranges.

This specification addresses only expansion buses with a physical address size matching the system physical address size. This ensures that an adapter on that bus can generate any valid physical system address. For support of private and hidden adapters, the host provides a region of host memory and I/O space that is considered private to the IOP. All other ranges are considered public. The IOP programs the ATU so it can claim any address in the public range and reasserts it unmodified on the system bus. Addresses in the private range do not need to be claimed by the ATU and, in fact, are expected to be claimed by private adapters on the expansion bus. The IOP's Bus/Local Access unit is an example of such device.

For this direct system access to function properly:

- The public memory space must be known.

- The offset (linear address translation) between each address in the public space and the address in the expansion bus space must be zero.
- Private adapters on the expansion bus may not be assigned memory addresses in the public range. Those adapters must be assigned private addresses.
- The expansion bus must have a private address segment. The IOP configures its private adapters to reside in a private address segment.
- Addresses in the private range are not claimed by the ATU.

2.5.3.6 Expansion/Local Access Unit

As mentioned previously, bus master adapters must be able to directly transfer data in and out of the IOP's local memory. This is illustrated by bus/local access unit 3x (Figure 2-17). This address translation unit captures memory cycles on the expansion bus within a particular address range and translates them to a memory access transactions on the IOP's local bus. This is exactly the same operation as the bus/system access unit, with two exceptions: First, the IOP establishes a single memory region within the private space for the local access unit, while the system access unit claims the set of public address ranges. A non-zero base offset may result. Second, the range of addresses on the expansion bus that the bus/local access unit claims must reside in the private address segment, and not be assigned to any adapter on that bus.

The IOP programs this unit with an address window that directly translates to the shared memory partition. The DDM needs to know the offset between the expansion bus address and the memory partition so it can properly program the bus master's DMA unit (i.e., program the bus master with the result of subtracting the offset from the local memory address).

2.5.4 Software Components

This section discusses the software components of an IOP.

2.5.4.1 IRTOS - I_2O Real Time Operating System

An IOP is required only to conform to the core specification (Chapter 5) and thus, the formal presence of an IRTOS, if the IOP supports loadable DDMs. Supporting loadable DDMs makes an IOP extensible. Even IOPs that do not support additional physical adapters should support loadable DDMs. In particular, this allows downloading ISMs. ISMs provide a mechanism for class extensions and other advancements, without modifying the original DDM.

2.5.4.2 IOP Memory Regions

This specification categorizes the IOP's local memory by its access capabilities. There are actually five classes of memory that overlap.

1. **IOP Private Memory**: This memory is accessed only by the IOP's processor. It is typically used for code and driver workspace.
2. **Bus-Accessible Memory**: This local memory is also accessible by bus masters on a particular bus. It is characterized as a single contiguous block containing a local and a bus base address. A particular memory location is identified as an offset from that base address. This region is important to an HDM if it creates data buffers that its adapters access. The HDM allocates

memory accessible to a particular adapter by identifying the bus where that adapter is located. Typically, this is the same as all local adapter memory.

3. **All Local Adapter Accessible Memory**: An HDM may control adapters on more than one bus and need them to share data buffers. Therefore, in one region, all bus accessible memory regions overlap. This is considered *all local adapter accessible* memory.

4. **All Adapter Accessible Memory**: An ISM does not necessarily know which adapters will access the data buffers it specifies in a message. If the ISM makes an external connection, the buffers that it specifies in its messages must accessible from the system bus. But even if the ISM makes only local connections, the DDM it claimed may use a remote connection. Thus an ISM needs to allocate memory accessible by any target device, even when it is on another IOP. Therefore, the ISM allocates memory that is accessible by all local adapters and other IOPs. This region is the *all adapter accessible memory* region.
   The actual need for system access to the all-adapter memory region rests with two criteria: If the IOP supports controlling adapters that reside on the system bus, or if it supports external connections, then the all adapter region must be accessible from the system bus. Figure 2-19 illustrates how overlapping access capabilities map to respective memory categories based on those policies.

5. **System Shared Memory**: At a minimum, a region of memory accessible via the system bus contains the inbound message frames where the host and other IOPs deposit their messages. Since hardware/software implementations vary, there may be no relationship between the system memory address for an inbound message frame and the location of that message in local memory when it is posted to a DDM's event queue (because the IOP or its hardware might copy the message). Additional system shared memory may be available to support peer connections. In this case, the DDM needs to allocate memory accessible from the system.

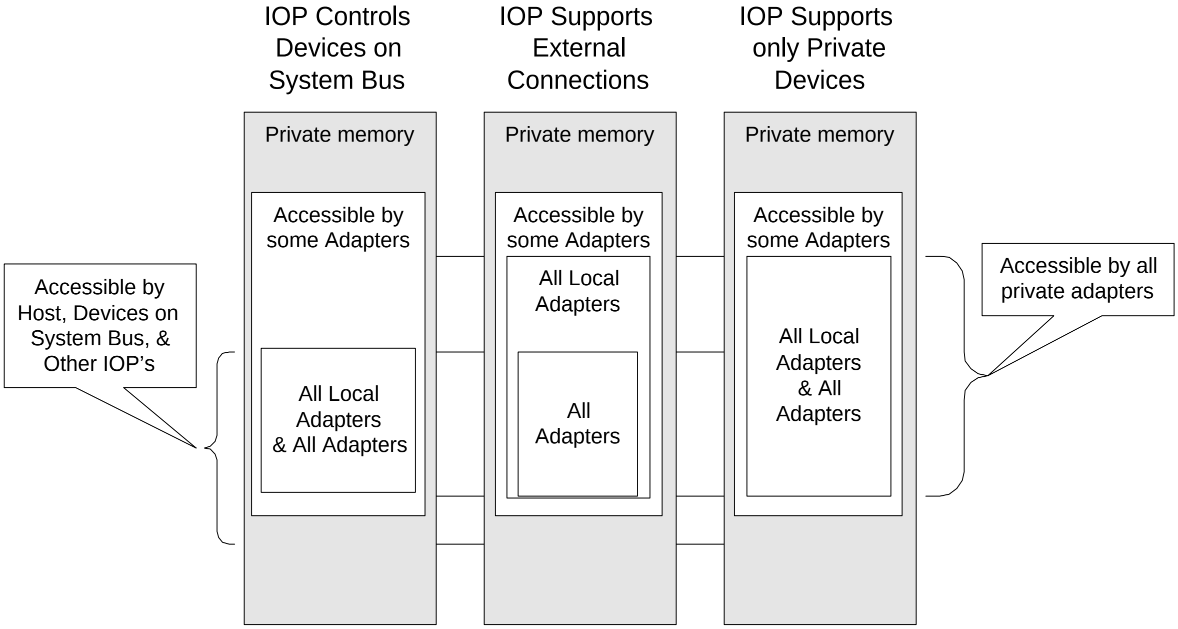

IOP Controls Devices on System Bus
IOP Supports External Connections
IOP Supports only Private Devices
Private memory
Private memory
Private memory
Accessible by some Adapters
Accessible by some Adapters
Accessible by some Adapters
All Local Adapters & All Adapters
All Local Adapters
All Adapters
All Local Adapters & All Adapters
Accessible by Host, Devices on System Bus, & Other IOP's
Accessible by all private adapters

Figure 2-19 IOP Memory Regions

Even though adapters on all buses may access all-local-adapter-accessible memory, they do not necessarily do so using the same physical address. The system bus and each I/O Expansion Bus has its own ATU that maps the adapter accessible memory to a region in that bus' address space. Thus, adapter-accessible memory has the following attributes:

Local Bus Address. The address the DDM uses to access the memory.

Adapter Bus Address. The address the adapter uses to access the memory. Each bus that can access a memory location has its own bus address value. The relationship between memory locations is linear. That is, the offset between any two memory locations is constant for all buses.

The core interface (i.e., DDM operation) is concerned with allocating memory in:

- the private memory region, for data structures the DDM does not wish to share with or make accessible to adapters.
- the adapter-accessible region, for data structures the HDM needs to make accessible to a single adapter (e.g., buffer descriptor lists and buffers for data cache).
- the all-local-adapter region, for the HDM that needs to share data structures between its adapters.
- the all-adapter region, for data structures the ISM needs to make accessible to its claimed devices (i.e., local buffers specified in the SGL of messages it sends).

The shell interface deals with the shared system memory region, since it represents the total memory accessible to the host and other IOPs.

IOP Processor Caching: Data caching is a popular approach to increasing performance, but one must assure data integrity. Many processor caching schemes do not provide cache coherent protocols that invalidate the processor's cache when adapters or DMA engines modify memory. Nor do they flush the processor's cache to memory before adapters access the data. Instruction caching is not an issue, but several issues regarding data caching follow:

- The IOP may cache private memory. The DDM should never use private memory for the object of a DMA request.
- Memory allocated as adapter accessible must be cache coherent. That is, it should not be cached unless the processor and memory support cache coherent protocols. This applies to memory allocated as adapter specific, all local adapters, and all adapters. Even when cache coherent protocols exist, processor caching for this memory region is often non-productive.
- Message frames: Since processing message frames is a prime operation, it is arguable whether messages need to be cacheable. However, they could contain data that needs to be accessed by adapters and therefore need to be in all-adapter memory. An IOP may elect write-through caching for message frames if it can assure that posting the message does not violate caching integrity. The DDM must always take the following precautions:
  - Messages are treated as read-only by adapter hardware. This allows the adapter to DMA the immediate data directly from the message and allows the DDM to specify its contents for the source of a DMA operation.
  - Hardware may never write to the message frame.

- Memory allocated as shared system memory must be cache coherent. That is, it should not be cached unless the processor and memory controller support cache-coherent protocols.

2.5.4.3 Bus Objects

All buses have resources associated with them, such as memory, adapters, and DMA controllers. A DDM running on the IOP directly accesses these resources. A bus object describes the resources on a bus and provides methods to access them.

The IOP's HAL creates a bus object for each expansion bus. The DDM learns of a bus object when it is assigned a physical adapter. The BusID is an attribute of the adapter (i.e., adapter object) and thus the DDM learns the BusID by querying the IOP about a particular adapter object.

2.5.4.4 DMA Objects

DMA objects are created by a DDM to move data between memory and adapters on the same or different buses, including the IOP's local bus. The IRTOS determines its available facilities and invokes them as it sees fit, providing the requested transfer function. The DDM creates a DMA object by specifying the source bus and cycle type (memory or I/O port), the target bus and cycle type. The DDM engages the DMA object by providing the specific source and destination addresses.

2.5.4.5 Adapter Objects

Depending on the characteristics of the expansion bus, a DDM learns adapter information in various ways. Typically, if the IOP's HAL knows of an adapter (as is possible with a PCI bus), the IRTOS assigns the adapter to the DDM indicating its AdapterID. From the AdapterID, the DDM learns the BusID and physical address. Other bus types may provide the DDM with only the information about the bus and let the DDM probe the private space for appropriate adapters. In either case, with the BusID, the DDM can allocate memory from the shared memory partition for that bus, create a DMA object to move blocks of data from that bus to another, and access the adapter.

2.5.4.6 API Transport Functions

Besides the DMA and bus objects, the IRTOS provides API functions that facilitate data movement:

- **Translate**. The DDM specifies a primary bus, an address on that bus, and a secondary bus. If an ATU can access the first bus from the second, the IRTOS provides the address on the second bus that accesses the specified location on the first bus.

- **Single memory access**. A set of functions for accessing a byte, 16-bit word, 32-bit word, or 64-bit word on a particular bus. For reading, the DDM specifies the bus and memory address. The IRTOS returns the value stored at that location. For writing, the DDM specifies the bus, the memory address, and data value. The IRTOS writes the value to that location.

- **Single I/O access**. A set of functions for accessing a byte, 16-bit port, 32-bit port, or 64-bit port on a particular bus. For input, the DDM specifies the bus and port address. The IRTOS returns the value read from that location. For output, the DDM specifies the bus, the port address, and data value. The IRTOS writes the value to that port.

3
Basic Requirements

This chapter discusses the basic requirements for implementing each piece of the I_2O system:

- the software on a host platform
- the hardware and software on an I/O platform
- and the device driver modules.

It also includes the facilities and structures common to I_2O interfaces.

3.1 Host Requirements

On the host platform, the I_2O components are the OSMs and the message layer.

3.1.1 System BIOS

In a system with an I_2O-aware OS, the BIOS (or its extensions) does not need to be I_2O aware unless it boots the OS from an I_2O device. However, a BIOS that is I_2O aware allows an OS that is not I_2O aware to access I_2O devices. In this instance, the BIOS (or its extensions) must abstract the I_2O subsystem to the OS and provide I_2O functionality via its normal BIOS function calls.

Each hardware or system vendor that develops an IOP that can provide service via a BIOS function call (e.g., hard disk via int 13) may wish to provide a BIOS extension; it can capture that function and register it with the BIOS when the system's BIOS is not I_2O aware. Also, hardware vendors who provide DDMs for adapters may also wish to provide a BIOS extension to capture that function and register it with the BIOS when either the system's BIOS is not I_2O-aware, or the adapter is not assigned to an IOP.

3.1.1.1 BIOS extensions for I_2O intelligent adapters (i.e., IOPs)

An intelligent I_2O device may provide a BIOS extension that initializes it in a system where the system's BIOS is not I_2O-aware. When the system's BIOS is I_2O-aware, the BIOS and its extension cannot both initialize an IOP. Thus, the BIOS extension must detect when the system's BIOS has initialized the IOP and, if so, yield to the system's BIOS. The BIOS extension can determine the presence of an I_2O-aware system BIOS by sending an ***ExecStatusGet*** request to the IOP. If the IOP returns any state except *Reset*, then an I_2O-aware system BIOS exists and has initialized the IOP.

3.1.1.2 BIOS extensions for other adapters

An adapter may provide a BIOS extension that initializes the adapter in a system without an I_2O-aware BIOS, or when an IOP does not control that adapter. Since the adapter vendor provides both the BIOS extension and the DDM, those modules must synchronize control of the adapter so that the two drivers do not compete or replicate service. An I_2O-aware BIOS can determine which adapters the IOP controls via the IOP's Hardware Resource Table and must either provide a means to expose that information to the BIOS extension, or prevent BIOS extensions from accessing those adapters. The I_2O architecture supports hidden devices to promote this requirement.

When the BIOS (or its extension) provides access to an I_2O device, it must update the BiosInfo field in the IOP's Logical Configuration Table to indicate the relationship between the device and the BIOS function. In addition, if the BIOS boots from an I_2O device, it must set the BootDevice field in the IOP's Logical Configuration Table to indicate the device used to boot the OS. The IOP preserves this information during the system transition from BIOS to OS.

3.1.2 Host Messenger Instance

The host OS provides a number of OSMs and the message layer. In addition to the normal message transport function, the OS provides the following I_2O system functions:

- An executive function that initializes and maintains the I_2O system.
- The system resource manager that configures and manages connections between IOPs.
- The configuration function that provides the user interface, file system access, and the configuration dialogue with an IOP and its DDMs. This function enables installing and configuring the IOP and its DDMs.

In addition, the OS provides the capacity to install OSMs produced by third-party vendors. The OSM interface is not specified by this document, but must provide the following:

- The ability to query the messenger for the list of IOPs and their registered devices (i.e., logical configuration table information).
- The ability to send requests and receive replies.

The OS hardware abstraction layer must prevent any legacy driver from locating an adapter listed in an IOP's hardware resource table as controlled by that IOP.

3.1.3 OS-Specific Modules

The interface and operating environment for an OSM are specified by the OS. The OSM must adhere to the message requirements specified in this chapter and in Chapter 6.

OSMs only send requests and receive replies. They neither send replies nor receive requests. OSMs do not need to establish connections, but they do need to claim devices they intend to consume.

The OSM must send only messages specified for the class for which the target is registered. The OSM must be capable of processing replies from the message layer as well as replies from its intended target. The OSM must be able to correlate replies with the appropriate request, based on the content of the Transaction Context field.

The OSM sees only one memory domain -- system memory -- and is not concerned with translating addresses. An OSM must be able to convert virtual addresses to physical addresses.

3.2 I/O Platform Requirements

The requirements for an IOP vary, depending on whether it supports loadable DDMs.

3.2.1 Private Platforms

A hardware or system vendor supplying an I_2O adapter (e.g., an intelligent adapter card providing both the IOP and embedded controllers), without supporting third party DDMs, must adhere to the requirements for the shell interface (Chapter 4) and the message requirements in this chapter and Chapter 6. Although the device need not implement the core interface (Chapter 5), it must function externally as if it does. When responding to an installation or load request for a DDM, the IOP rejects the request as function not supported (see 3.4.1.2.4).

3.2.2 Open Platforms

A hardware or system vendor supplying an I_2O subsystem (e.g., an IOP on the motherboard) that can support third party DDMs must adhere to the requirements for the shell interface (Chapter 4), the core interface (Chapter 5), and the message requirements in this chapter and Chapter 6. Features that differentiate between designs include the amount of non-volatile memory for storing third party DDMs, as well as the physical expansion bus capability.

3.2.3 IOP Design Considerations

The IOP provides an execution environment for its modules, and thus provides the same capabilities as an operating system. The physical resources are the CPU, local data and code memory, non-volatile storage for drivers and parameter blocks, timer functions, and DMA capability. In addition, the IOP must provide a physical messaging unit. The amount, type, and location of non-volatile storage is not specified, but left to the ingenuity of the developer.

The effectiveness of an IOP depends not only on its processing performance and capability, but also on its transport mechanism. This pertains to both message and data transporting. Message transportation copies data (the message) into the target's memory (the message frame) and notifies the target (message queues).

DMA capabilities must exist to transfer data between the local and system memory in both directions. If the IOP supports expansion buses, it must support transferring data between I/O adapters and local memory. The IOP must also provide access to any configuration cycles needed on its internal buses and for adapters on the system I/O bus that it controls.

If the IOP supports controlling adapters on a system bus, it must be able to route interrupts so that the adapter does not generate interrupts to the host.

The IOP may provide other capabilities, such as battery-backed RAM for data caches and non-volatile memory. Such features are strictly at the discretion of the vendor.

3.3 Device Driver Module Requirements

The requirements for a DDM vary, depending on whether it is a loadable driver.

3.3.1 General Requirements

All drivers must be able to receive and reply to requests as specified in this chapter and in Chapter 6. The driver must support all utility and base class messages defined for the driver's registered class.

3.3.2 Loadable Driver Modules

In addition to the general requirements, a loadable driver must adhere to the core interface and behavior as specified in Chapter 5. The driver must be fully able to operate given the facilities defined in this specification.

3.4 Common Facilities and Structures

The rest of this chapter discusses the facilities and structures common to the core, shell, and message-based interfaces.

3.4.1 Message Structure and Definitions

The I_2O components in a system communicate by exchanging messages. The messages are data structures containing: a fixed-size header and a variable-size payload. These two parts reside within a physically-contiguous buffer called the message frame, shown in Figure 3-1. The message frame is allocated in shared memory.

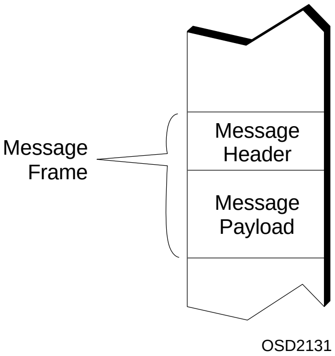


Figure 3-1. Message Frame

Messages fall into two basic categories:

- *Request* messages initiate activity at the destination. A request can contain multiple transactions of the same type.
- *Reply* messages return status information concerning one or more requests.

Generally, every request has an associated reply that concludes the transaction. The ratio of replies to requests need not be one-to-one; each request can have multiple replies or a single reply can have multiple requests. Certain requests do not evoke a reply. Such exceptions are explicitly identified in the message definition.

Requests and replies share the same basic structure. The difference between a request and a reply is that the request payload may contain a scatter-gather list (SGL) and the reply may contain a transaction reply list (TRL).

3.4.1.1 Message Header

Message headers are fixed in size, structure, and location within the message frame. They convey two types of information: device addressing and payload description. The addressing information is used primarily by the routing mechanism within the transport layer to properly deliver the message. The function code is used primarily by the target to determine the structure of the payload section. Figure 3-2 shows the structure of a message header:

<table>
<tr><th colspan="2">31 3 24</th><th colspan="2">23 2 16</th><th colspan="2">15 1 8</th><th colspan="2">7 0 0</th><th></th></tr>
<tr><td colspan="4">MessageSize</td><td colspan="2">MessageFlags</td><td colspan="2">VersionOffset</td><td>0</td></tr>
<tr><td colspan="2">Function</td><td colspan="3">InitiatorAddress</td><td colspan="3">TargetAddress</td><td>4</td></tr>
<tr><td colspan="8">InitiatorContext</td><td>8</td></tr>
</table>

Figure 3-2. Message Header Version 001

Fields

Function — A function code that identifies the requested action as well as the structure of the message payload (i.e., the parameters). See Section 3.4.1.2.4 for details on Function ranges. In a reply, the Function code indicates the action being reported as well as the structure of the reply payload.

InitiatorAddress — An IOP-unique identifier of the initiator of the request.

InitiatorContext — Initiator–private item. The initiator sets this field. The target must always return this value unchanged in the reply message. The initiator's message layer typically uses this value to identify the handler for the reply. The size of this field is either 32 bits or 64 bits, as identified by the MessageFlags. This specification provides the ability for both the IOP and DDM to function in an environment that supports both context sizes. This is a concept that has not been verified or tested. Unless the DDM indicates that it supports both context sizes concurrently, it may assume that context size is a constant and it does not need to test each message.

MessageSize — Total number of 32-bit words occupied by the message including the header. Although it is intended primarily for use by the transport layer to optimize the physical data movement process, this field can be useful in transports that do not inherently track the actual size of the message frame.

Since MessageSize includes the header, the minimum legal value for a MessageSize is 3 (12 bytes).

MessageFlags — Provides status information about the message, as follows:

Bit 0: Static - A 0 identifies a normal message and a 1 identifies a static message frame.

Bit 1: ContextSize - A 0 identifies 32-bit context field sizes (Initiator Context and Transaction Context) and a one identifies 64-bit context field sizes.

Bits 2-3: reserved

Bit 4: Multiple - A 0 identifies a single transaction message and a 1 identifies a multiple transaction message. When this value is zero, the

Transaction Context is the first field in the message payload. When it is one, then each Transaction Context is specified in the SGL for a request and in the TRL for a reply.

Bit 5: Fail - Message processing failure: A 0 identifies normal delivery and a 1 indicates that the message could not be processed. This bit indicates that the original request was not processed and that the structure of this reply is a ***FaultNotification*** message (see 3.4.1.2.3). Note that this bit indicates transport failure and must not be set to indicate transaction failure.

Bit 6: Last - In a request, this bit is reserved. In a reply, a 1 indicates the last reply in a multi-reply transaction. This bit pertains to the Transaction Context conveyed in the reply. The notion of multiple replies per transaction is established on a class basis and depends on the function code. Some functions within a class support multiple replies per transaction (e.g., status reports), and others do not. This bit must be set in single reply messages, even though the last reply is the only one that is ever generated. This bit is specific to the transaction context: a zero indicates a progress report; a one concludes each transaction identified in the message and releases all associated buffers.

Bit 7: Reply - A 0 identifies the message as a request and a 1 identifies it as a reply.

TargetAddress A simple integer identifier of the recipient of the request. Target addresses are unique within the domain of an IOP. This is the logical address of the intended recipient of the request. The transport layer uses it to look up the characteristics of the target, such as the event handler, preferred method of delivery, and message priority.

VersionOffset Identifies the structure of the message header and provides the offset of the SGL/TRL, as follows.

Bits 0-2: MessageHeaderVersion: Identifies the structure of the message header. The message header in Figure 3-2 is identified by MessageHeaderVersion 001b. All IOPs that claim compliance with this specification must support version 001b message headers. In future releases of this specification, new message headers may not adhere to the current header structure. These message headers will have a higher version number, allowing the recipient of the message, as well as all the intermediate layers in the delivery chain, to correctly interpret the structure of the header.

Header version 001b is a little endian format that accommodates both 32-bit and 64-bit operating systems. Future versions might define big endian headers.

Bit 3: reserved

Bits 4-7: SglTrlOffset - A four-bit field that indicates the location of the SGL or TRL within the message frame. This field contains the offset (in number of 32-bit words) of the start of the list from the start of the message header. If there is no list, the initiator sets this value to zero.

Note

The actual values for the TargetAddress *and* InitiatorAddress *fields are generated and maintained by the IOPs. ID=000h sent to an IOP identifies the IOP's I_2O executive and not a DDM within that IOP. ID=001h is reserved for the host operating system and should never be sent to an IOP as a target address. An IOP never assigns ID=001h as an ID of a DDM.*

3.4.1.2 Message Payload

Immediately following the message header, a section of variable size contains all additional information associated with the message. The structure of this section depends on the class of the message and the Function field. The content of this section conveys to the target the operational details (parameters) that define the requested action. For the reply, it conveys the status of the request along with any immediate data.

Typically, the payload of a request contains a TransactionContext field, which identifies the specific request and the target returns that TransactionContext value in a reply message. TransactionContext fields are treated the same as the InitiatorContext field. That is, the target copies the value from the request to the reply.

Although the message payload is not meant to transmit raw data, it can be desirable to do so. When the raw data is very small or when the same transport mechanism moves both, packaging the data with the message reduces the amount of activity on connecting buses.

Often, the payload for a request needs to refer to memory. If so, it requires a scatter-gather list (SGL). The SGL uses a standard format for memory references that is understood by the originator, the target, the transport, and all intermediate software layers.

Also, if a request message supports multiple transactions (Multiple bit set in MessageFlags) it requires an SGL. The SGL bundles details for each transaction and identifies it by its Transaction Context in a manner that the transport and all intermediate software layers understand.

The equivalent of an SGL for a reply message is the transaction reply list (TRL). If the reply concludes multiple transactions (Multiple bit set in MessageFlags) then it requires a TRL. The TRL bundles the details for each transaction and identifies it by its Transaction Context in a manner that both the transport and all intermediate software layers understand.

Due to its varying length, if an SGL or TRL is included in a message, it is the last structure in the payload, and thus follows any message details. Because the length of the message details varies by function, the start of the SGL or TRL is identified in the message header so it can be located without knowing the message class.

3.4.1.2.1 Request Message Structures

The template for a *single transaction request* message is shown in Figure 3-3.

<table>
<tr><th colspan="2">31 3 24</th><th colspan="2">23 2 16</th><th colspan="2">15 1 8</th><th colspan="2">7 0 0</th><th></th></tr>
<tr><td colspan="4">MessageSize</td><td><b>0</b> 0 0 <b>0</b></td><td>x x x x</td><td colspan="2">VersionOffset</td><td>0</td></tr>
<tr><td colspan="2">Function</td><td colspan="3">InitiatorAddress</td><td colspan="3">TargetAddress</td><td>4</td></tr>
<tr><td colspan="8">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="8">TransactionContext</td><td>12 (16)</td></tr>
<tr><td colspan="8">MessageDetails
(Function Specific)</td><td>16 (24)</td></tr>
<tr><td colspan="8">SGL</td><td>n</td></tr>
</table>

Offset in () signifies offset for 64-bit context fields

Figure 3-3. Single Transaction Request Message Template

Fields

Function — Identifies the purpose of the message and the structure of its details. See the Function field in section 3.4.1.1.

InitiatorAddress — The TID of the requesting module.

InitiatorContext — An arbitrary value assigned by the message layer of the requesting module. This value is returned in the reply and is used to route the reply to the appropriate message handler.

MessageDetails — Provides detailed information about the specific request. Each message class defines the size and content of this field, which varies based on the value of the Function field.

SGL — As defined in section 3.4.2. Typically, the SGL identifies the source data and/or the reply buffers.

TargetAddress — The TID of the module that receives the message.

TransactionContext — An arbitrary value assigned by the initiator. This value is returned in a reply and the requester typically uses it to correlate the reply to the original request.

VersionOffset — As defined in 3.4.1.1. The SglTrlOffset in the VersionOffset field is set to zero when there is no SGL, and to n/4 when the SGL is included (where n is the offset of the SGL).

The template for a *multiple transaction request* is in Figure 3-4.

31 3 24	23 2 16	15				1			8	7 0 0	
MessageSize		**0**	0	0	**1**	x	x	x	x	VersionOffset	0
Function	InitiatorAddress					TargetAddress					4
InitiatorContext											8
MessageDetails (Function Specific)											12 (16)
SGL											n
TransactionInfo 1											
TransactionInfo *n*											

Offset in () signifies offset for 64-bit context fields

Figure 3-4. Multiple Transaction Request Message Template

In lieu of a single Transaction Context field, the SGL contains a list of transaction details where each set of details specifies a Transaction Context and its respective data buffers (as defined in 3.4.1.1). The SglTrlOffset in the VersionOffset field is set to n/4.

3.4.1.2.2 Normal Reply Message Structures

The template for a normal single transaction reply is shown in Figure 3-5. A normal reply is defined as a message with the Reply bit set and the Fail bit reset.

31 3 24	23 2 16	15				1			8	7 0 0	
MessageSize		**1**	x	0	**0**	x	x	x	x	VersionOffset	0
Function	InitiatorAddress					TargetAddress					4
InitiatorContext											8
TransactionContext											12 (16)
ReqStatus	reserved	DetailedStatusCode									16 (24)
ReplyPayload (Function and Status specific)											20 (28)

Offset in () signifies offset for 64-bit context fields

Figure 3-5. Single Transaction Reply Message Template

Fields

DetailedStatusCode — This field accommodates a more detailed status when required. Values for this field are defined by the particular message class and Function. Detailed status codes for Executive Class, DDM Class, Utility Class, and Transaction Error replies are specified in Table 3-2.

Function — The value copied from the Function field of the request.

InitiatorAddress — The value from the InitiatorAddress field of the request.

InitiatorContext — The value copied from the InitiatorContext field of the request.

ReplyPayload — Provides more detailed information as required. Each message class defines the size and content of this field, which vary based on the Function and ReqStatus codes.

ReqStatus — This field conveys the general status of the transaction per Table 3-1.

TargetAddress — The value from the TargetAddress field of the request.

TransactionContext — The value copied from the TransactionContext field of the request.

VersionOffset — As defined in section 3.4.1.1. The SglTrlOffset in the VersionOffset field is set to zero.

Table 3-1. Reply Status Codes

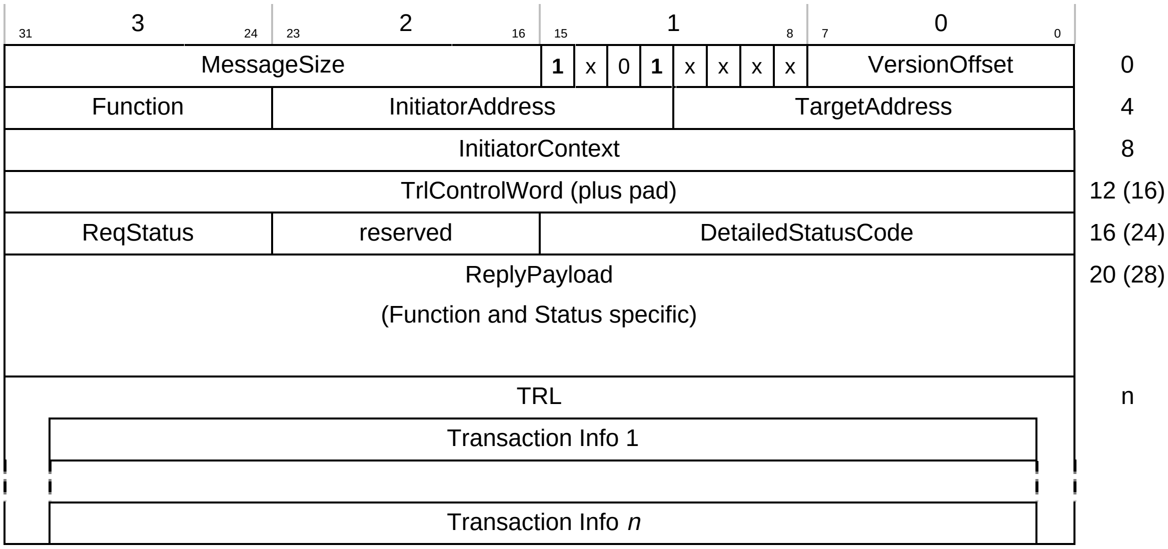
ReqStatus (I2O_*REPLY_STATUS_*xxx)	**Description**
_SUCCESS	Normal completion without reportable errors. The DetailedStatusCode reports any warning or residual status information.
_ABORT_DIRTY	Aborted by originator – cannot conclude abort
_ABORT_NO_DATA_TRANSFER	Aborted by originator – no data transfer
_ABORT_PARTIAL_TRANSFER	Aborted by originator – partial completion
_ERROR_DIRTY	Error in execution – cannot conclude completion
_ERROR_NO_DATA_TRANSFER	Error in execution – no data transfer
_ERROR_PARTIAL_TRANSFER	Error in execution – partial completion
_PROCESS_ABORT_DIRTY	Aborted due to system command or reconfiguration – cannot conclude completion
_PROCESS_ABORT_NO_DATA_TRANSFER	Aborted due to system command or reconfiguration – no data transfer
_PROCESS_ABORT_PARTIAL_TRANSFER	Aborted due to system command or reconfiguration – partial completion
_PROGRESS_REPORT	Progress Report (FINAL bit set to 0)
_ TRANSACTION_ERROR	The DDM or IOP can not process request (see 3.4.1.2.4)

Note: *_ABORT_DIRTY*, *_ERROR_DIRTY* and *_PROCESS_ABORT DIRTY* indicates hardware programmed with the address of the buffer and that the target cannot guarantee that the hardware will not access that buffer.

Table 3-2. Detailed Status Codes

ReqStatus (I2O*_DETAIL_STATUS_*xxx*)*	**Description**
__SUCCESS	Normal completion without reportable errors.
_BAD_KEY	The specified key was not recognized or invalid. (Applies only to operations on table groups.)
_CHAIN_BUFFER_TOO_LARGE	The SGL Chain Buffer is too large to be processed.
_DEVICE_BUSY	Device is busy with another operation and its request queue is full.
_DEVICE_LOCKED	Resource locked – resource exclusively reserved by another requester (see ***UtilLock*** message and ***UtilDeviceReserve*** message)
_DEVICE_NOT_AVAILABLE	Device can not be accessed via this TID.
_DEVICE_RESET	Resource reset – not available until ***UtilResetAck*** received
_INAPPROPRIATE_FUNCTION	This function is not valid for this class or sub-class.
_INSUFFICIENT_RESOURCE_HARD	Insufficient resources available to process the message. Retrying the same message is not advised.
_INSUFFICIENT_RESOURCE_SOFT	Insufficient resources are available to process the message. This situation may be temporary, so retrying the same message could succeed.
_INVALID_INITIATOR_ADDRESS	Invalid InitiatorAddress
_INVALID_MESSAGE_FLAGS	Invalid MessageFlags field value
_INVALID_OFFSET	Invalid SGL/TRL offset value in message header
_INVALID_PARAMETER	Invalid parameter
_INVALID_REQUEST	Invalid request – resource not allocated to requester
_INVALID_TARGET_ADDRESS	Invalid TargetAddress
_MESSAGE_TOO_LARGE	Message too large – MessageSize specifies a value larger than the message frame
_MESSAGE_TOO_SMALL	Message too small – MessageSize less than the minimum allowed
_MISSING_PARAMETER	Missing parameter
_NO_SUCH_PAGE	The requested page was not found in the device's script table. The reply buffer contains valid HTML describing the error.
_REPLY_BUFFER_FULL	The reply overflowed the reply buffer (or reply message frame). The reply buffer contains data generated up to the point of overflow.
_TCL_ERROR	The TCL interpreter reported an error while processing the TCL script for the requested page. The reply buffer contains valid HTML describing the error.
_TIMEOUT	Service or device did not respond within the allocated time.
_UNKNOWN_ERROR	An error condition not covered by any other code occurred.
_UNKNOWN_FUNCTION	Unknown Function code
_UNSUPPORTED_FUNCTION	The requested function is not supported.
_UNSUPPORTED_VERSION	Invalid/unsupported message header Version field value

The template for a normal *multiple transaction reply* is in Figure 3-6. All transactions reported by a multiple transaction reply must be for the same function. This structure is optimized for reporting a number of successful transactions.



3 (31–24)	2 (23–16)	1 (15–8)	0 (7–0)	
MessageSize		**1** x 0 **1** x x x x	VersionOffset	0
Function	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TrlControlWord (plus pad)				12 (16)
ReqStatus	reserved	DetailedStatusCode		16 (24)
ReplyPayload (Function and Status specific)				20 (28)
TRL Transaction Info 1 Transaction Info *n*				n

Offset in () signifies offset for 64-bit context fields

Figure 3-6. Multiple Transaction Reply Message Template

Additional Fields

TrlControlWord — The TRL Control Word (see 3.4.3) replaces the single Transaction Context value and identifies the format of the TRL that follows the reply payload. When Transaction Context size is 64 bits, 32 bits of pad follow the TRL Control Word, placing the status word at the same offset for both single and multiple transaction replies.

TRL — The TRL provides a list of transaction contexts and their details (see 3.4.3). Each class defines the size and content of transaction details, which may vary based on the Function and status codes.

VersionOffset — As defined in section 3.4.1.1. The SglTrlOffset in the VersionOffset field is set to n/4 where n is the offset of the TRL.

3.4.1.2.3 Fault Reply Message Structure

Two mechanisms convey message failure to the initiator of the message. When a request cannot be processed, a ***FaultNotification*** reply returns to the initiator of the failed request, as specified below. When a reply cannot be processed, there is no mechanism to reply to a failed reply, so a ***UtilReplyFaultNotify*** request message must be created and sent to the device that sent the failed reply, as specified in Chapter 6.

Figure 3-7 specifies the reply resulting from message failure when the message layer cannot deliver the request to the target, or the target cannot process the request. The module that detects the failure generates the ***FaultNotification*** reply. Following the standard header, a status block details why the message could not be delivered. Following the status block is the MFA of a buffer containing the original message. Since the DDM accesses the message via local memory, the IOP's messenger translates the local address to a valid MFA (copying the message if necessary).

The ***FaultNotification*** applies to all classes of messages. A failure reply contains both the Reply and Fail bits set in the MessageFlags field (see section 3.4.1.1). The Final bit is also set in a failed reply because there will be no other replies.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		1 1 1 0 x x x x	VersionOffset	0
ORIGFUNCCODE	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext (Null)				12 (16)
FailureCode	Severity	HighestVersion	LowestVersion	16 (24)
FailingHostUnitID		reserved	FailingIOP_ID	20 (28)
AgeLimit				24 (32)
PreservedMessage(Low 32 bits)				28 (36)
PreservedMessage(High 32 bits)				32 (38)

Offset in () signifies offset for 64-bit context fields

Figure 3-7. Reply Message for Message Failure

Field

AgeLimit The maximum number of microseconds that the transport allows to elapse when trying to deliver the message. A value of FFFFh indicates no time limit. At present, aging of messages in the transport layer is not defined.

FailingHostUnitID HostUnitID for the failing unit.

FailingIOP IOP_ID of the failing IOP.

FailureCode Table 3-3 shows the possible FailureCode values.

Function *ORIGFUNCCODE* = the value of the Function filed in the request message.

IOP_ID The IOP_ID (as assigned by the host using the ***ExecSysTabSet*** message) for the messenger that rejects the message.

LowestVersion The lowest I$_2$O version that the rejecting module supports.

HighestVersion The highest I$_2$O version that the rejecting module supports. Currently 01h.

PreservedMessage

This field provides a pointer to the original message, which is preserved. It is the offset address of a message frame that holds (a copy of) the original message. This message frame must be associated with the IOP's inbound message queue. A DDM rejecting a message specifies the message handle of the corrupted message and the IOP converts it to an inbound MFA. The originator must resubmit, reuse, or release the message frame containing the preserved message. The message frame can be released by changing the Function value to *NOP* and resubmitting it.

Note:

The Message Frame Address (MFA) is an offset that the host (or IOP) writes to the inbound Message FIFO. A DDM rejecting the message can only supply the message handle that the IOP provided to the DDM. Therefore, the IOP must convert that message handle to an MFA of a frame that contains the original message. The IOP may use the actual message frame of the original message, or copy the original message to a new frame, but the IOP must know that the originator of the original message will reuse that message frame.

Severity Indicates the severity of the failure.

Bit 0 FormatError – this message can never be delivered/processed.
Bit 1 PathError – this message can no longer be delivered/processed.
Bit 2 PathState – the system state does not allow delivery.
Bit 3 Congestion – resources temporarily not available; do not retry immediately.

TransactionContext Set to zero. This is a place holder.

Table 3-3 Message Failure Codes

Code	Description
81h	Transport service suspended to the specified target
82h	Transport service terminated for the specified target
83h	Transport congestion – Temporary lack of transport resources to complete transaction
84h	Transport failure -- could not deliver message
85h	Transport state prevents delivery
86h	Time out – could not post message within the appropriate age limit
87h	General routing failure – cannot forward frame to intended IOP
88h	Invalid/unsupported message header Version field value
89h	Invalid SGL/TRL offset value
8Ah	Invalid MessageFlags field value
8Bh	Message too small – MessageSize less than the minimum allowed
8Ch	Message too large – MessageSize specifies a value larger than the message frame
8Dh	Invalid TargetAddress
8Eh	Invalid InitiatorAddress
8Fh	Invalid InitiatorContext
FFh	Unknown transport failure

3.4.1.2.4 Transaction Error Reply Message

When an IOP or DDM rejects a message for general cause (format error, bad function code, insufficient resources, etc.), the target returns a reply message with a ReqStatus value of *I2O_REPLY_STATUS_ TRANSACTION_ERROR* as shown in Figure 3-8. The reply is a single transaction reply. If the request message was a multiple transaction request, the error reply is repeated for each transaction that the target rejects.

The ***TransactionErrorReply*** applies to all classes of messages. An error reply contains the Reply bit set to one but the Fail bit set to zero (in the MessageFlags field see section 3.4.1.1). The Final bit is set appropriately.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		1 x 0 0 0 0 x 0	VersionOffset	0
Function	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
TRANSACTION_ERROR	reserved	DetailedStatusCode		16 (24)
ErrorOffset				
reserved			BitOffset	

Offset in () signifies offset for 64-bit context fields

Figure 3-8. Reply Message for Transaction Error

Field

ErrorOffset Indicates the byte location in the message frame where the error was detected (the VersionOffset field = 0000h). This value is set to zero if the error is not associated with a particular field, such as for lack of resource availability.

DetailedStatusCode See Table 3-2.

BitOffset Indicates the bit location (0 through 7) within the byte indicated by ErrorOffset. This value is typically zero except when the errored field is a flags field or a bit-specific field. This value is set to zero if the error is not associated with a particular bit.

3.4.1.3 Function Codes

Messages within a certain class are further divided into types. The Function field in the header identifies the type of message, as defined in the Table 3-4.

Table 3-4. Relationship Between Message Type and Function Field

Function Value		
From	**To**	**Type**
00h	1Fh	Utility (common to all message classes)
20h	0FEh	Base (unique to each message class)
0FFh	0FFh	Private (allows vendors to add value)

All drivers compliant with the I_2O specification must support the utility and base type messages. Support for private messages is optional and must be negotiated while establishing a link.

3.4.1.4 Utility Messages

A common set of utility messages applies to every class. These messages provide the most basic functionality to enable negotiation and configuration. All components that claim compliance with this specification must support all the utility messages defined in Chapter 6. Initiators assume each target supports all utility messages and that support is not subject to negotiation.

3.4.1.5 Base Messages

Base messages typically describe I/O requests or service requests. Again, all components that claim compliance with this specification must support all the base messages within the registered class. Initiators assume that each target supports all base messages within the target's registered class and that support is not subject to negotiation.

3.4.1.6 Private Messages

Private messages allow extensions of the base set of messages within a class, to support new functionality without creating a new class.

All messages that have a Function field value of *Private Message* are private extensions to base class messages. Private messages allow vendors to add value without having to create a new class.

The private message includes, in addition to the transaction information, a field that uniquely identifies the organization defining the extended codes and a field specifying the extended function code identifying the specific function.

The structure of the PrivatePayload of a private message is not described by this specification. That structure is defined by the vendor that created the extension, as identified by the OrganizationID field.

Figure 3-9 shows the structure of a private type request message.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
Function = 0FFh	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
OrganizationID		XFunctionCode		16 (24)
PrivatePayload defined by owner of OrganizationID specific for XFunctionCode				20 (28)

Figure 3-9. Private Type Request Message Template

Fields

OrganizationID — Contains the 16-bit ID assigned by the I_2O Special Interest Group to the organization defining the message. The value 0000h is reserved for extensions assigned by the I_2O SIG. The SIG assigns each member company an OrganizationID, allowing each company to define their own set of message extensions.

TransactionContext — Contains the transaction context if the Multiple bit in the MessageFlags field is not set. When Multiple is set, the content of this field is not defined. The size of this field depends on the value of the ContextSize bit in the MessageFlags.

XFunctionCode — Function code extension. This value is administered by the organization specified by the OrganizationID field and identifies the structure for the remainder of the message payload.

3.4.2 Addressing Memory (Scatter-Gather Lists)

Modules pass data to each other by providing a structured list of memory addresses that specifies data buffers and their respective lengths. The memory of a data buffer can be scattered, rather than contiguous, and allocated as page frames. This specification defines a standard format for a scatter-gather list (SGL). The SGL can indicate any number of buffers, each of which can contain any number of segments (fragments). This specification uses the term data buffer to mean memory shared between the initiator and target. The information passed from the initiator to the target must explicitly define the data buffers involved in each transaction.

A transaction consists of the following components:

- Zero or more data buffers that contain the source data
- Zero or more data buffers where the results of the transaction are placed
- Additional parameters providing details about the specific transaction

A request message provides information about one or more transactions, where each transaction contains any number of these components. The SGL provides a structured method for indicating each component.

In general, the request message payload contains details of all the included transactions (*request details)* followed by an SGL. The SGL provides the information for each transaction (*transaction details)*. There are two models for requests: single and multiple transaction.

- The *single transaction request model* does not differentiate between message details and transaction parameters. Therefore, transaction parameters typically reside in the message detail portion of the payload and the SGL provides a list of source data and reply buffers. Nothing precludes transaction parameters from being included in the SGL except the following:

  The transaction context is a critical parameter. It is an arbitrary value, supplied by the initiator of a request, that correlates the data buffers of a transaction with its disposition (see section 3.4.3). For single transaction requests, this field must be in a known location independent of the message class or function. Therefore, it is always the first field in the message payload. This differs from the multiple transaction request that supplies the transaction context in the SGL.

- The *multiple transaction request model* groups details for each transaction. In addition, the first component of a transaction specifies the context for that group. By definition, a transaction set consists of all components starting with the one that specifies the transaction context and includes all following components, until another transaction context is specified. The high level structure is shown in Figure 3-10.

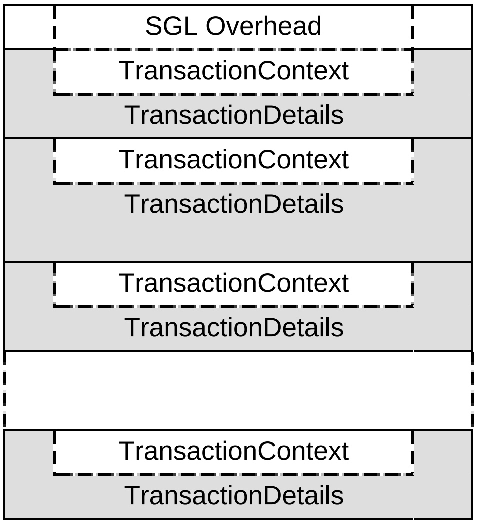


Figure 3-10 High Level Structure of a Multiple Transaction SGL

The SGL specifies source data by identifying a buffer, which may contain multiple segments. A large segment of a data is generally indicated by its location in common memory. This provides efficiency since just the data's location is passed between functions and only the terminating function moves the data. For smaller data segments, the data itself may be supplied in the SGL (immediate data). This is useful when intermediate functions need the content of that segment, or when the size of the data is not much larger than the size of the structures specifying the buffer.

The SGL is a structured list of elements, each of which provides specific detail. Some element types identify a segment of a data buffer, while others provide transaction parameters or information about the SGL.

A buffer is one or more segments of memory containing a logically contiguous data structure. The SGL provides three methods of describing the segments that compose a buffer.

1. *Simple addressing*: A single fragment represented by a physical address and a length.
2. *Page frame addressing*: Fixed length fragments (*data page frames* or *pages*) indicated in sequence. Data may start anywhere in the first page listed and end anywhere in the last page listed. More details of page addressing are provided later.
3. *Immediate data*: Data embedded in the SGL element.

Each segment is either a physically contiguous block of memory or a page frame list. A buffer is described in the SGL by a consecutive list of data elements, each specifying a segment of the buffer. The last element is marked as the end of the buffer. All segments for a particular buffer are listed in order.

Note: The SGL supports 64-bit physical addressing. This provides compatibility with future versions. This version of the specification requires only support for 32-bit addresses. In addition, this version of the specification defines both 32- and 64-bit context fields. The size of the context fields is an environmental parameter. Even though these definitions allow a single binary to support both sizes, it is not required. DDMs and IOPs indicate their ability to function with the various context sizes and are never loaded into an environment with a context size they do not support. Unless the DDM indicates that it supports both context sizes concurrently, it may assume that context size is a constant and it does not need to test each message.

The SGL is structured to satisfy the following requirements:

- Physical address field size is an attribute of the I/O subsystem and is either 32 or 64 bits. Both system and local address use the same size address fields in the SGL.
- The format for a 64-bit address field places the low order address bits in the low order byte locations. That way, a 32-bit address can be placed in a 64-bit address field by padding the high order four bytes with zeros.
- The size of context fields for a particular SGL is constant.
- The Initiator Context field is the same size as the Transaction Context field and is either 32-bit or 64-bit. Buffer Context fields are 0, 1, 2, or 3 times the size of the Transaction Context field.

3.4.2.1 SGL Format

The following figure shows the structure of a typical SGL. The structure consists of variable length elements. The most significant bit of the first 32-bit word in each element is the LastElement flag, which identifies the last element in the sequence. This specification also discusses a chain pointer indicating a buffer containing an additional sequence of elements. The pointer appends logically to the end of the original sequence.

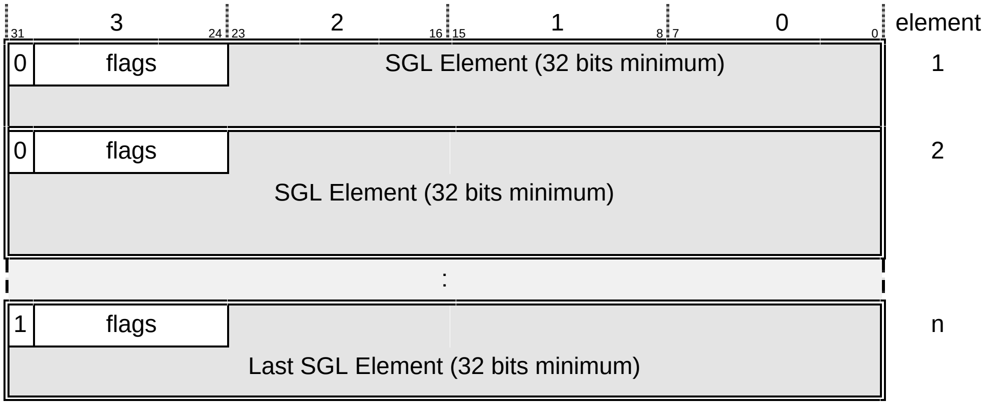

3
2
1
0
element
31
24
23
16
15
8
7
0
0
flags
SGL Element (32 bits minimum)
1
0
flags
SGL Element (32 bits minimum)
2
:
1
flags
Last SGL Element (32 bits minimum)
n

Figure 3-11. Physical Structure of a SGL Sequence

Figure 3-12 illustrates how various element types may be arranged to form an SGL. This illustration contains *m* number of elements: *n* are present in the message and *m-n* are in the additional list elsewhere.

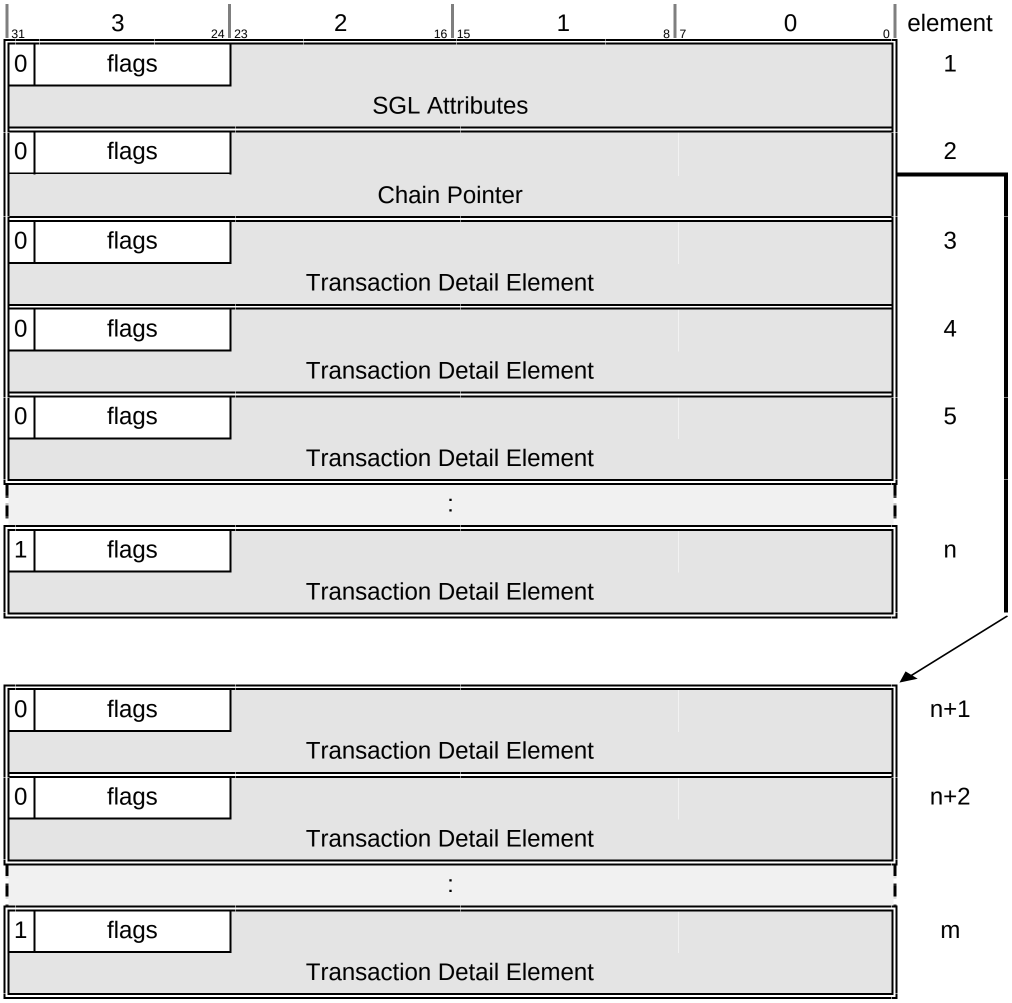

3
2
1
0
31
24
23
16
15
8
7
0
element
0
flags
SGL Attributes
1
0
flags
Chain Pointer
2
0
flags
Transaction Detail Element
3
0
flags
Transaction Detail Element
4
0
flags
Transaction Detail Element
5
:
1
flags
Transaction Detail Element
n
0
flags
Transaction Detail Element
n+1
0
flags
Transaction Detail Element
n+2
:
1
flags
Transaction Detail Element
m

Figure 3-12. Logical Structure of a SGL

The first element in the sequence is reserved for SGL attributes. The content of the SGL Attributes element is specified later in this section. The SGL Attributes element is necessary only if the attributes of the SGL differ from the default. SGL attributes include address size, context size, and page frame size. If the SGL Attributes element is absent, the default attributes are used. When it is present, it is always the first element. An entire SGL contains, at most, a single SGL Attributes element.

The next element in the order is reserved for an SGL chain pointer. The SGL Chain Pointer element provides the size and location of the buffer containing the remainder of the SGL. The SGL Chain Pointer element is specified later in this section. If the SGL Chain Pointer element is absent, the SGL is a single sequence of elements with all elements present in the message frame. When an SGL Chain Pointer is present, it is always the first or second element. An entire SGL contains, at most, a single SGL Chain Pointer element.

The remainder of the SGL contains zero or more sets of transaction elements. A transaction set contains any number of elements specifying data buffers and transaction parameters. The following rules apply:

- Memory addresses and transaction contexts may appear in a message only as expressly specified.
- The number of components (buffers and transaction parameter elements) in a transaction depends on class and function.
- The content and interpretation of data and parameters are specific to class and function.
- All elements that define a buffer must be consecutive (excluding chaining). Transaction parameter elements may precede, follow, or intermix with buffers, as long as elements within each buffer are contiguous.
- A data buffer is one or more segments of memory that either:
  — contains the source data to be operated on, or
  — stores the results of the operation (reply buffer).
- A buffer that both provides source data and is the reply buffer must be listed twice: first as a source buffer, and second, as a reply buffer, both in the same transaction set.
- All messages must use either the single transaction model, where the transaction context is the first field in the message payload, or the multiple transaction model, whose first element specifies the transaction context. All subsequent buffers, until a new transaction context is specified, belong to that transaction.
- The life of a buffer is from the time the request message is generated until a FINAL reply message, containing that transaction context, is returned.
- Ignore elements may appear anywhere in the SGL.

The SGL starts in the message frame and may continue in a separate physically contiguous buffer, called the SGL Chain Buffer. Designating the SGL Chain Buffer at the beginning of the SGL allows the parsing module to set up a DMA request to retrieve the remainder of the SGL without having to parse through it. While the target retrieves the SGL Chain Buffer, it processes the SGL elements located in the message itself. When more than one SGL Chain Buffer is specified, the SGL Chain Pointers must appear in the appropriate order, but need not be densely packed. In fact, a number of factors determine the optimum placement of the chain pointers. For this version of the specification, *only one SGL Chain Buffer is allowed*. This is a requirement for modules creating or modifying the SGL. Modules parsing the SGL *may* reject messages with more than one SGL Chain Buffer specified and *must* reject those with more SGL Chain Buffers than it can handle.

3.4.2.2 SGL Element Formats

SGL elements are always a multiple of 32 bits long. Each element conforms to one of the following formats. These definitions provide the foundation for all SGL elements. The class definition in Chapter 6 specifies which transaction element types are appropriate and their order in the SGL.

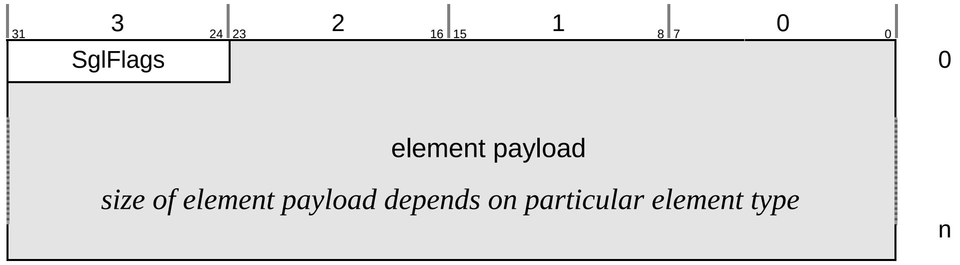

31
3
24
23
2
16
15
1
8
7
0
0
SglFlags
0
element payload
size of element payload depends on particular element type
n

Figure 3-13 Compressed Format

The *compressed* format provides density since there is only one byte of overhead. Buffer segment descriptors use this format, which achieves the maximum amount of buffer description in the minimum space. For this format, the element length is different for each type. The definitions later in this section identify each element type using this format and provide the respective formulas for determining the element's length. This is the format used extensively in version 1.0.

Because determining element length depends on a priori knowledge, future definition of element types using this format presents an interoperability issue: Drivers cannot determine the size of unknown elements, and therefore cannot find the next element. The definition of such an element type requires either locating the element at the end of the list (LE bit set), or including it only in messages with a higher MessageHeaderVersion in the message header.

All remaining formats expressly include element length as a field allowing new definitions without severe impact. When a driver encounters an unknown element type, it must ignore it, skipping to the next element.

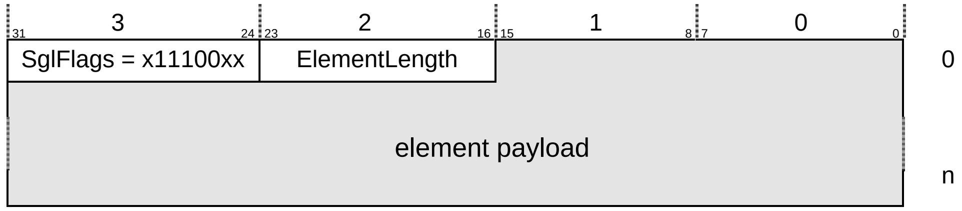


Figure 3-14 Short Element of Specified Length

The *short element* format is defined for small amounts of information. The ElementLength field specifies the length of the element in 32-bit words. Therefore, the size of the element may vary from four to 1020 bytes. An ElementLength value of zero is not allowed.

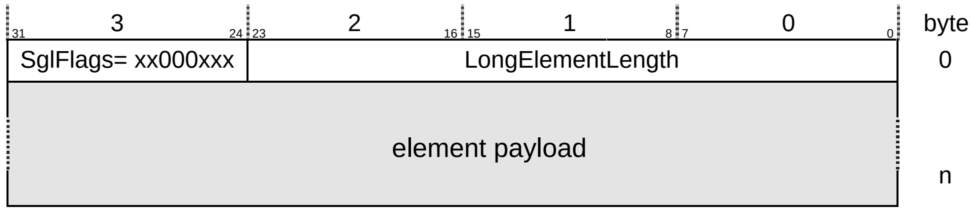


Figure 3-15 Long Element of Specified Length

The *long element* format allows large amounts of information. The LongElementLength field specifies the length of the element in 32-bit words. A LongElementLength value of zero is not allowed. Therefore, the size of the element may vary from four to 67,108,860 bytes.

The SglFlags values assigned to this format coincide with Ignore entries defined in version 1.0. Thus, these elements are simply ignored by drivers implemented using the 1.0 specification.

3.4.2.2.1 SglFlags

Bits 24 through 31 of the first 32-bit word in each SGL element contain the SglFlags field. The SglFlags field identifies the type of element, defined in Table 3-5. Bit 31 (bit 7 of the SglFlags) is the Last Element (LE) bit. When the module parsing the SGL reaches an element with this bit set, it looks for an additional SGL Chain Buffer. If one exists, then parsing resumes from the beginning of the SGL chain buffer, after parsing the element with the LE bit set. If no chain buffer exists, then this is the last element in the SGL and parsing is complete.

Table 3-5 SglFlags Field Definition

		SGL Flags Values								
Element Description	**v1.0**	**b7**	**b6**	**b5**	**b4**	**b3**	**b2**	**b1**	**b0**	**Element Size**
reserved for long format		LE	x	**0**	**0**	0	x	x	x	4 to 64 Mbytes
Ignore Element	✓	LE	0	**0**	**0**	0	0	**0**	**0**	“
Transport (reserved)	[note1]	LE	0	**0**	**0**	0	1	**0**	**0**	“
Long Transaction Parameters	[note1]	LE	1	**0**	**0**	0	0	bc1	bc0	“
Bit Bucket		LE	eob	**0**	**0**	1	0	bc1	bc0	4 to 20 bytes
Immediate Data Segment		LE	eob	**0**	**0**	1	1	bc1	bc0	4 to 16 Mbytes
Simple Address Segment *[note2]	✓*	LE	**eob**	**0**	**1**	LA*	dir	**bc1**	**bc0**	8 to 24 bytes
Page List Address Segment *[note2]	✓*	LE	**eob**	**1**	**0**	LA*	dir	**bc1**	**bc0**	8 to 128K bytes
Chain Pointer*[note2]	✓*	LE	0	**1**	**1**	LA*	0	0	0	8 to 12 bytes
Chain Pointer w/context		LE	0	**1**	**1**	LA*	0	bc1	bc0	8 to 24 bytes
reserved for short format		LE	1	**1**	**1**	x	x	x	x	4 to 1020 bytes
Short Transaction Parameters		LE	1	**1**	**1**	0	0	bc1	bc0	“
SGL Attributes		LE	1	**1**	**1**	1	1	0	0	“

All other values for SglFlags are reserved for future use and may not appear in a message with MessageHeaderVersion = 001b.

The 1.0 column identifies elements defined in version 1.0:

[note1] This element is an ignore element in version 1.0.

[note2] The LA bit was only 0 (system address) in version 1.0.

* = The LA bit must be 0 for 1.0 compatibility.

bc1, bc0 impact element length.

Definitions:

bc1, bc0 = buffer context field size as shown in Table 3-6. A 00b indicates a buffer context is not present. When present, the Buffer Context field starts at offset 4. The buffer context field may contain multiple fields, the first of which is reserved for a Transaction Context (see rules below).

dir = identifies the direction for the data buffer:

0 = input buffer (I/O places data in buffer).

1 = output data (I/O reads from the buffer).

eob = end of buffer flag. A 1 indicates end of the buffer and thus the next data element (if any) starts a new buffer.

LA = identifies if the address is an IOP local address (1) or a node address (0).

LE = last element flag. When this bit is set, this is the last element in the sequence, but not necessarily the end of the list.

x = any value.

3.4.2.2.2 Physical Address and Buffer Context fields

Many elements contain a physical address field. For 32-bit addressing, the size of this field is four bytes, but may increase in future versions that accommodate 64-bit physical addressing. For this version, the size of address fields is constant. A DDM identified as a *32-bit-only driver* may expect every address to be 32 bits long.

Local addresses (LA bit set) are valid only between DDMs on the same IOP. SGLs generated by DDMs may include local addresses. When an SGL is sent to a remote DDM, the IRTOS must convert all local addresses to system addresses. The SGL Attributes element indicates the presence or absence of local addresses. Therefore, a DDM creating an SGL without local addresses should include the SGL Attributes element if it sends the message to a remote DDM. This allows the IRTOS to quickly determine that the SGL does not require repair.

Buffer Context Rules: Many of the elements contain a buffer context field. Its size varies and it may hold a transaction context and/or class-specific information. The following rules apply:

1. The size of the Buffer Context field depends on the size of Transaction Context field (TcSize). See Table 3-6. TcSize is an attribute of the SGL and is constrained to either 32 bits or 64 bits. If TcSize is not explicitly specified via an SGL Attributes element, then it defaults to the context size of the message.

Table 3-6 Buffer Context Field Size

bc1	bc0	TcSize	Buffer Context field size (BcSize):
0	0	n/a	0 - Buffer Context field is absent
0	1	32 bits	32 bits (4 bytes)
1	0	32 bits	64 bits (8 bytes)
1	1	32 bits	96 bits (12 bytes)
0	1	64 bits	64 bits (8 bytes)
1	0	64 bits	128 bits (16 bytes)
1	1	64 bits	192 bits (24 bytes)

2. If a transaction parameter element (i.e. short or long transaction parameter) contains a Buffer Context field, then it is the first element of a transaction, and the Buffer Context field contains a Transaction Context as its first entry. If the Buffer Context field's length exceeds a Transaction Context, then the remainder of the field is class and function specific. This means that a transaction parameter element does not contain a Buffer Context fields unless it is the very first element of the transaction.

3. If the first segment of a buffer element (i.e., simple address, page list, immediate data, or bit bucket) contains a Buffer Context field, it defines the buffer as the first of a transaction and the Buffer Context field contains a Transaction Context as its first entry. If the Buffer Context field is larger than a Transaction Context, then the remainder of the field is class and function specific.

4. If a buffer segment other than its first contains a Buffer Context field, then the Buffer Context field does not contain a Transaction Context and all information in the Buffer Context is class and function specific.

5. Naturally, elements that do not contain a Buffer Context field do not contain a Transaction Context. Thus a transaction context must only appear in a request message as either the first field of the message payload, or in the first field of the Buffer Context of the first buffer segment or transaction parameter element of each transaction.

3.4.2.3 SGL Element Definitions

The details for each element follow, listed in alphabetical order by element name.

3.4.2.3.1 Bit Bucket Element **(new)**

A Bit Bucket element describes a segment of data that does not need to be transferred. Since there is no need to transfer the data, the target should ignore the transfer. If not, it should provide a trash buffer close to the transferring adapter to minimize the burden on the system. The result of any following operation is the same as if the Bit Bucket element were replaced with a simple address element of the same ByteCount value. The structure for the Bit Bucket element is shown in Figure 3-16.

3								2	1	0	Byte
31							24	23 16	15 8	7 0	
LE	eob	0	0	1	0	bc1	bc0	ByteCount			0
BufferContext											4

Figure 3-16. Bit Bucket Element Structure

Fields

BufferContext Variable size field as defined above. The size of this field is specified by the bc1::bc0 bits. See *Buffer Context Rules* in section 3.4.2.2.1.

ByteCount The size (in bytes) of the data segment to skip or dump.

SglFlags Bit 31 - LE bit;
Bit 30 - End of Buffer. A 1 indicates the last segment of data comprising the buffer;
Bit 29::26 - a value of 0010b identifies this as a Bit Bucket element;
Bit 25::24 - size of Buffer Context field; see Table 3-6.

The size of this element depends on the size of the BufferContext field, as follows:

element length = BcSize + 4 bytes

3.4.2.3.2 Chain Pointer

SGL chaining provides a method for including an SGL that does not fit in the message frame. The chain pointer element describes a buffer containing the next segment of the SGL (SGL Chain Buffer). Although the chain pointer element appears before the end of the current segment, the SGL Chain Buffer is actually processed as if it appeared at the end of the current segment.

3								2	1	0	Byte
31							24	23 16	15 8	7 0	
LE	0	1	1	LA	0	bc1	bc0	ByteCount			0
BufferContext											4
Physical Address											

Figure 3-17. Chain Pointer Structure

Fields

BufferContext Variable size field as defined above. The size of this field is specified by the bc1::bc0 bits. See Buffer Context rules in section 3.4.2.2.1. When present, this field contains the Chain Context. Currently, there is no definition or requirement for a Chain Context.

ByteCount — The size of the buffer in bytes that contains the next portion of the SGL.

PhysicalAddress — The address of the first element in the SGL chain buffer. The size of this field is 32 bits for this version of the specification.

SglFlags — Bit 31 - LE bit;
Bit 30::26 -a value of 011x0b identifies this as a Chain Pointer element;
Bit 27 indicates that the address is a system address (0) or an IOP address (1)
Bit 25::24 - size of Buffer Context field, see Table 3-6.

The size of this element depends on the address size (determined by the LA bit) and the size of the BufferContext field, as follows:

element length = BcSize + AddrSize + 4 bytes,

where AddrSize is the address size and BcSize is described in Table 3-6.

Note: A final reply containing the last transaction context in the message implicitly releases the SGL chain buffer.

An illustration of a SGL with a chain pointer is shown in Figure 3-18. The last element in the first sequence has its LE bit set to 1, as does the last element in the extended list.

Chaining is useful when the entire SGL cannot be packed into the message frame. Allowing a mixture of internal and external lists reduces the latencies on certain data transfer operations, because the first few addressing elements are available within the message frame. The driver can operate on them while it fetches the remainder of the list.

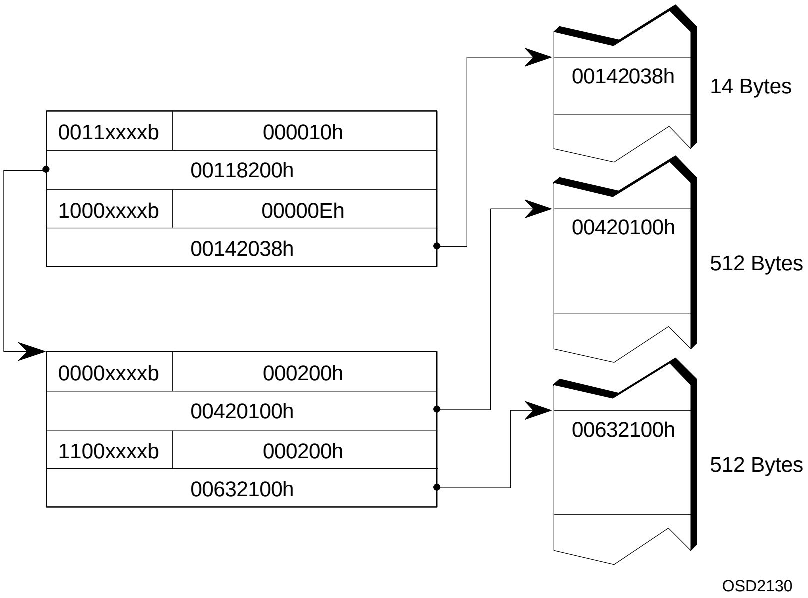
A chain list element can be the only element in the initial list. In this case, the entire list is supplied by the SGL chain buffer.

Example

To address a 1038-byte buffer made up of the three segments shown in Table 3-7, using a chained addressing mode, the SGL might look like Figure 3-18.

Table 3-7. Three Segments in a Buffer

Segment	Size (bytes)	Physical Address
1	14	00142038h
2	512	00420100h
3	512	00632100h



{ed. note: change drawing - SglFlags= 00110000, 10010000, 00010000, 11010000}

Figure 3-18. Chain Addressing Mode Example

3.4.2.3.3 Ignore Element

An *ignore element* contains no valid information. These elements must be skipped when processing the list and may be deleted from the list as it is transported or copied. The WordCount in such an element specifies the number of 32-bit words to skip. The minimum is one, because the count includes the first word. Some uses of this element type are: null SGL, place holders, and fill. Its structure is shown below.

31			3				24	23 2 16	15 1 8	7 0 0	Byte
LE	0	0	0	0	0	0	0	WordCount (32-bit words)			0
additional words to skip											

Figure 3-19. Ignore Element

Field

WordCount The number of 32-bit words to skip. The minimum is one, because the count includes the first word.

3.4.2.3.4 Immediate Data Element

An *immediate data element* describes a segment of data that is contained in the element. This element is equivalent to a simple address element that contains the address of the DATA field in the SGL, except there is no address to change when the SGL is moved or copied. The structure for the immediate data element is shown in Figure 3-20.

While a pointer to a data buffer is efficient for a large segment of data, the immediate data element is useful for a small amount of data.

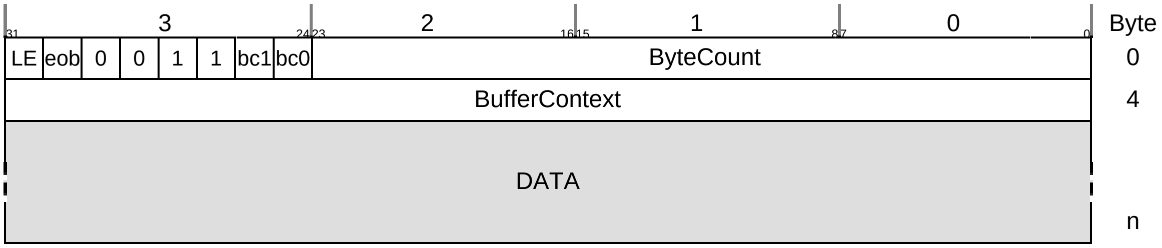


Figure 3-20. Immediate Data Element Structure

Fields

BufferContext — Variable size field as defined above. The size of this field is specified by the bc1::bc0 bits. See *Buffer Context Rules* in section 3.4.2.2.1.

ByteCount — The size (in bytes) of the DATA. This value does not include the size of the SglFlags, ByteCount, or BufferContext fields.

DATA — The exact number of bytes of data is specified in ByteCount. The next element starts on the first 32-bit boundary after the last byte of DATA.

SglFlags — Bit 31 - LE bit;
Bit 30 - a 1 indicates the last segment of data comprising the buffer;
Bit 29::26 - a value of 0011b identifies this as an immediate data element;
Bit 25::24 - size of Buffer Context field (see Table 3-6).

The size of this element is determined by the ByteCount field and the size of the buffer Context field, as follows:

element length = ByteCount + BcSize + 4 bytes + pad

Where **pad** is 0,1,2, or 3 bytes, to make the element length a multiple of four bytes, and **BcSize** is described in Table 3-6.

3.4.2.3.5 Long Transaction Parameters Element **(new)**

The *long transaction parameter* element conveys class-specific information associated with a particular data buffer or transaction. The information within the element is as shown in Figure 3-21.

Each class specification determines the content and relevance of such information, the structure of the Info field, and the location of the element within the SGL. The Info structure is class and function specific.

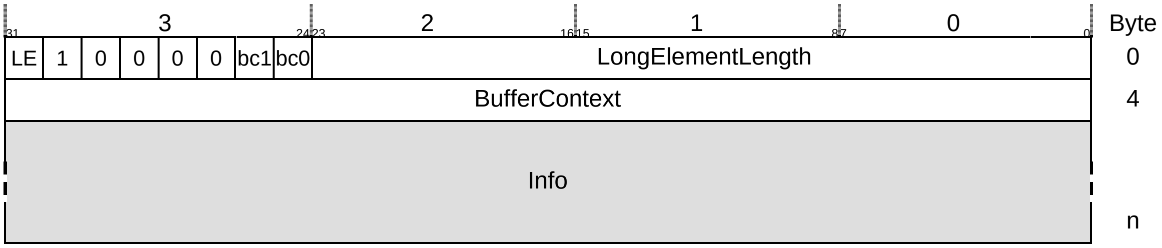


Figure 3-21. Long Transaction Parameters Element Structure

Fields

BufferContext Variable size field, as defined above. Its size is specified by the bc1::bc0 bits. See *Buffer Context Rules* in section 3.4.2.2.1. If this field exists, then it contains a transaction context.

Info Class-specific structure.

LongElementLength The size (in 32-bit words) of the entire element.

SglFlags Bit 31 - LE bit;
Bit 30:: 26 - a value of 10000b identifies a Long Transaction Parameters element;
Bit 25::24 - size of Buffer Context field (see Table 3-6)

3.4.2.3.6 Page List Addressing **(enhanced)**

The page list addressing mode addresses buffers that span multiple pages that are not contiguous. The first and last pages of a list can contain partial pages of data. That is, the buffer segment can start anywhere in the first page specified and end anywhere in the last page specified. The intermediate pages always contain exactly a page size of data. Where the data starts in the first page is specified by the first physical address in the page list. Where data ends in the last page is derived by subtracting the amount of data in the first and intermediate pages from the ByteCount and adding the result to the base address of the last page.

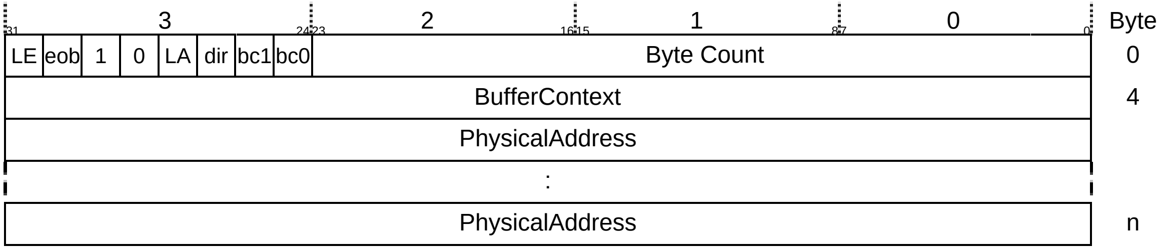


Figure 3-22. Page List Addressing Mode

Fields

ByteCount The total number of bytes addressed by this element.

BufferContext Variable size field as defined above. The size of this field is specified by the bc1::bc0 bits. See *Buffer Context Rules* in section 3.4.2.2.1.

PhysicalAddress

The first PhysicalAddress field contains the address of the first data byte. Any subsequent PhysicalAddress fields hold the address of pages that make up the rest of the buffer segment. For a four-kilobyte page size system, the lower 12 bits would be 0 in all but the first PhysicalAddress. The size of each PhysicalAddress field is 32 bits for this version of the specification.

SglFlags Bit 31 - LE bit;
Bit 30 - a 1 indicates the last segment of data comprising the buffer;
Bit 29::28 -a value of 10b identifies a Page List element;
Bit 27 indicates either a node address (0) or an IOP address (1);
Bit 26: a 0 indicates a reply buffer to be filled by the I/O transaction and a 1 indicates the buffer contains source data.
Bit 25::24 - size of Buffer Context field (see Table 3-6).

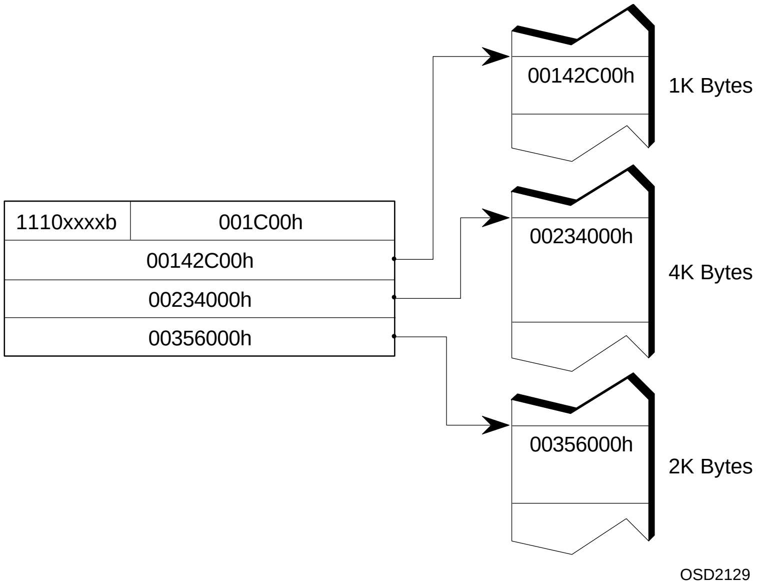
The local host establishes the default page size for the node when the host initializes the IOP. It can be any value from 256 to 16 Mbytes that is an integral power of two.

The size of this element depends on the size of the Buffer Context field, the address size, and number of the Physical Address fields, as follows.

element length = (NumberOfPages x AddrSize) + BcSize + 4 bytes
Where **NumberOfPages** is the number of pages frames needed, as described above; **AddrSize** is 32 bits for this version; and **BcSize** is described in Table 3-6.

Example

Suppose that a seven-kilobyte buffer spans three physically non-contiguous four-kilobyte pages (00142000h, 00234000h, and 00356000h) and starts at address 00142C00h. To specify the buffer using a page list addressing mode, the SGL should look like Figure 3-23.



{ed. note - change picture - SglFlags= 11100000 }

Figure 3-23. Page List Addressing Mode Example

3.4.2.3.7 SGL Attributes Element **(new)**

An *SGL attributes* element follows the structure in Figure 3-24. If this element is present, it must be the first in the SGL (other than an ignored element). This element establishes the address size, context size, and page frame size for the entire list. When this element is not present, the DDM uses the default values for those parameters. A DDM may reject messages with other than 32-bit addresses.

3								2	1	0	Byte
31							24	23 16	15 8	7 0	
LE	1	1	1	1	1	0	0	ElementLength	SglAttributeFlags		0
PageFrameSize											4
Additional parameters defined at a later date, ignored by this version											8

Figure 3-24. SGL Attributes Element

Field

ElementLength Specifies the number of 32-bit words in the entire element. The minimum is one, because the count includes the first word.

PageFrameSize Specifies the page frame size, in bytes, for this SGL. This value overrides the default value established for the node.

SglAttributeFlags Bit-specific flags identifying the attributes of this SGL.

Bits 15::11 reserved.

Bit 10 - Bit bucket hint - When set, this SGL contains no Bit Bucket elements.

Bit 9 - Immediate data hint - When set, this SGL contains no Immediate Data elements.

Bit 8 - Local Address hint - When set, this SGL contains no local addresses.

Bits 7::3 reserved

Bit 2 - Size for Transaction Context fields, 0=32 bits, 1=64 bits.

Bit 1 - reserved.

Bit 0 - Size for Local Address fields, 0=32 bits, 1=reserved.

SglFlags Identifies the SGL element:
Bit 31 - LE bit;
Bit 30::24 - a value of 1111100b identifies this as a SglAttribute element

3.4.2.3.8 Short Transaction Parameters Element **(new)**

The *short transaction parameter* element conveys class-specific information associated with a particular transaction that does not involve a memory reference. The information is contained within the element, as in Figure 3-25. This element is more efficient than the Long Transaction Parameters element for conveying small blocks of information, since it has only two bytes of overhead.

Each class specification determines the content and relevance of such information, the structure of the Info field, and the location of the element within the SGL. The Info structure is class and function specific.

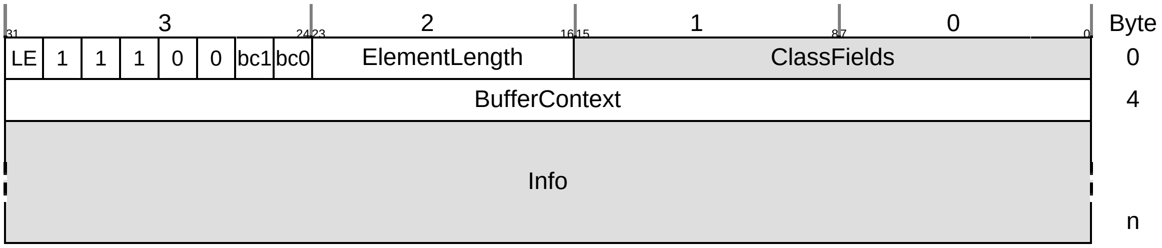


Figure 3-25. Short Transaction Parameters Structure

Fields

BufferContext — Variable size field as defined above. The size of this field is specified by the bc1::bc0 bits. See *Buffer Context Rules* in section 3.4.2.2.1.

ClassFields — Reserved for class-specific definition.

ElementLength — The size (in 32-bit words) of the element.

Info — The structure of the Info field is defined by each class by function and padded to fill the element.

SglFlags — Bit 31 - LE bit
Bit 30::26 - a value of 11100b identifies a Short Transaction Parameters element
Bit 25::24 - size of Buffer Context field (see Table 3-6)

3.4.2.3.9 Simple Addressing **(enhanced)**

The simple addressing mode shown below is for a single buffer segment that is contiguous in physical memory. It is also useful for quickly converting existing address/count type scatter-gather lists into an I_2O addressing scheme, with minimal modifications.

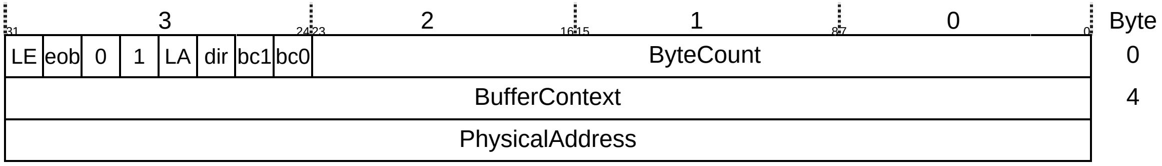


Figure 3-26. Simple Addressing Mode

Fields

BufferContext — Variable size field, as defined above. The size of this field is specified by the bc1::bc0 bits. See *Buffer Context Rules* in section 3.4.2.2.1.

ByteCount — The total number of bytes addressed by this element.

PhysicalAddress — The address of the first byte of data in this buffer segment. Its size is 32 bits for this version of the specification.

SglFlags Bit 31 - LE bit
Bit 30 - a 1 indicates this is the last segment of data comprising the buffer
Bit 29::28 -a value of 01b identifies this as a Simple Address element
Bit 27 indicates that the address is a node address (0) or an IOP address (1)
Bit 26: a 0 indicates this is an empty buffer to be filled by the I/O transaction and a 1 indicates

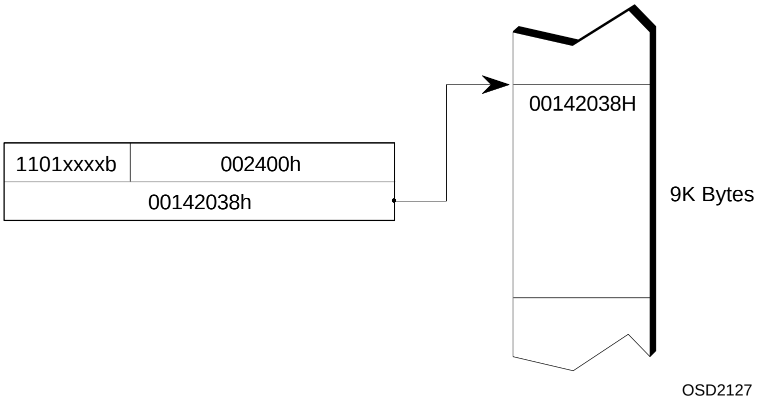
it is a full buffer to be consumed by the I/O transaction.
Bit 25::24 - size of Buffer Context field (see Table 3-6)

The size of this element depends on the sizes of the Buffer Context and the Physical Address fields, as follows.

element length = AddrSize + BcSize + 4 bytes
Where **AddrSize** is 32 bits and **BcSize** is described in Table 3-6.

Example 1

To address a physically-contiguous nine-kilobyte buffer that starts at address 00142038h, using simple addressing mode, the address list should look like Figure 3-27.

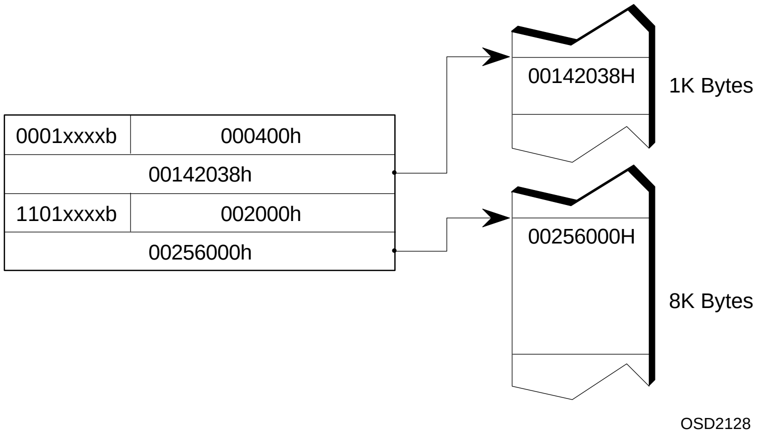


{ed. note - change picture - SglFlags= 11010000}

Figure 3-27. Simple Addressing Mode Example 1

Example 2

Suppose that the nine-kilobyte buffer is not physically contiguous, but contains a one-kilobyte section that starts at address 00142038h, and an eight-kilobyte section that starts at address 00265000h. To address the buffer using simple addressing mode, the address list should look like Figure 3-28.



{ed. note - change picture - SglFlags= 00010000, 11010000}

Figure 3-28. Simple Addressing Mode Example 2

3.4.2.3.10 Transport Detail Element **(new)**

The *transport detail* element is reserved for embedding transport-specific information in the SGL. DDMs always ignore such elements. The code point is reserved so that the transport layer can use this element type without conflict with future element definitions. The format of the element is shown in Figure 3-29.

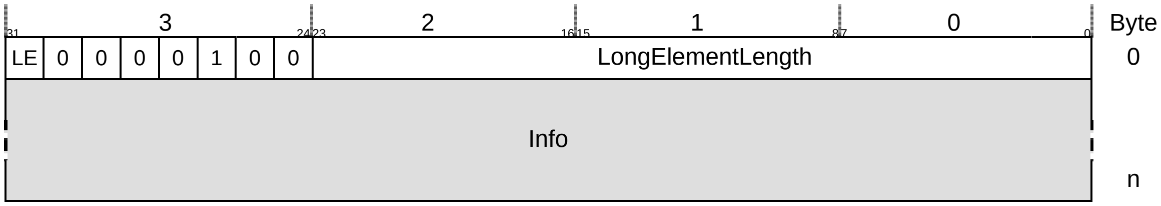


Figure 3-29. Transport Element Structure

Fields

Info Transport-specific structure.

LongElementLength The size (in 32-bit words) of the element.

SglFlags Bit 31 - LE bit;
Bit 30::24 - a value of 0000100 identifies a Transport Detail element

3.4.3 Buffer Management Styles

Every buffer is associated with a transaction context. The transaction context correlates zero, one, or a set of buffers with a transaction reference. When the initiator specifies a buffer in the SGL, its access privilege is temporarily passed to the target module. When the target module finishes the transaction, it supplies the transaction context with its appropriate status in a reply to the initiator. This reply indicates the conclusion of the transaction and thus returns ownership of the buffer.

In other words, generating each request conveys access rights to the target for each buffer listed. When the target concludes the transaction, it releases those rights by specifying the transaction context in a final reply (reply message with the FINAL bit set). The life of a transaction context (and thus the temporary ownership of the buffers) is from the time the request message is generated until the associated transaction context value is returned in a final reply. During this time, the target has access rights to those data buffers. After the final reply, the target has no access rights to those buffers.

This specification provides for intermediate status reports. In this case, the target conveys the transaction context in a reply, but the FINAL bit is not set. The originator should understand that an intermediate status report does not indicate the amount of data available in a reply buffer. In fact, where remote transports are involved, the target may operate on an intermediate buffer and the transport does not transfer the data until the target sends the final reply.

For remote transports between nodes, the transport mechanism tracks buffer assignments so it can set up the necessary mapping or allow shipping data between the nodes. The transport layer needs to know when it can release the mappings or when it must ship the results back. The transport accomplishes this by tracking the transaction context. Therefore, the context *must* appear only in known locations in both requests and replies.

3.4.3.1 Request Messages:

Two styles exist for specifying the transaction context in requests: the single transaction and the multiple transactions request models. Message class definitions in Chapter 6 determine the style for a particular message type.

The single transaction request model provides a single transaction context in a well-known location in the message payload. For this style, the transaction context is the first field in the message payload.

The multiple transactions request model supplies a number of transaction context values in the SGL; that is, one transaction context for each set of elements comprising a transaction.

The originator indicates the transaction style, the size of the transaction context field, and the location of the SGL in the message header.

3.4.3.2 Reply Messages

Two styles are also available for specifying the transaction context in replies: the single transaction reply model and the multiple transaction reply model. The style for a particular reply does not necessarily depend on the style of the request. Message class definitions in Chapter 6 determine the reply style for a particular message type.

The single transaction reply model provides a single transaction context in a well-known location in the message payload. For this style, the transaction context must be the first field in the message payload. The multiple transaction reply model provides a list of transaction contexts (the Transaction Reply List or TRL). The TRL includes transaction details (status) for each transaction context.

The multiple transaction reply model provides a TrlControlWord in a well-known location in the message payload. For this style, the TrlControlWord must be the first field in the message payload (where the TransactionContext is normally located in single transaction replies).

The target indicates the transaction style, the size of the transaction context field, and the location of the TRL in the message header of its reply.

3.4.3.3 Transaction Reply Lists (TRLs)

The general structure of a TRL (Figure 3-30) is a list of TransactionContext values with their associated details. The details can be in one of three formats:

1. single fixed length element
2. single variable length element
3. multiple fixed length elements.

The TrlControlWord indicates the particular format, as shown in Figure 3-31.

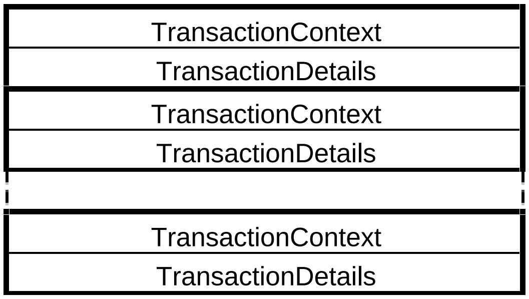


Figure 3-30 Transaction Reply List General Structure

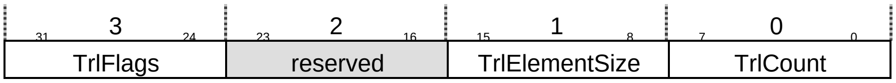


Figure 3-31. TRL Control Word

Fields

TrlCount — Total number of transactions in the TRL.

TrlElementSize — Size of each element (number of 32-bit words). For variable length elements, this value is zero.

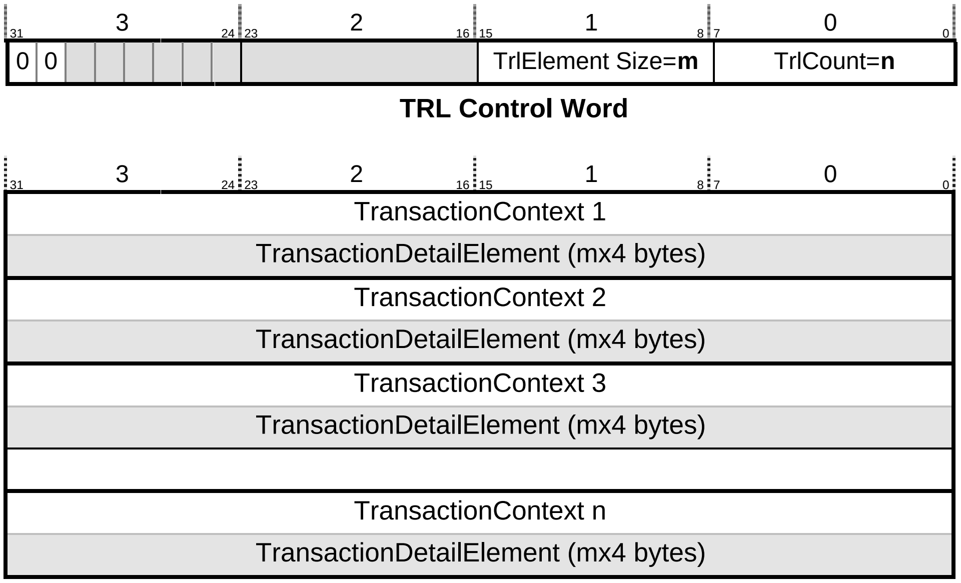
TrlFlags — Bits 31:30 identify the TRL transaction detail format as follows:

00 = Single fixed-length element (Figure 3-32)

01 = Single variable-length element (Figure 3-33)

10 = Multiple fixed-length elements (Figure 3-34)

11 = reserved

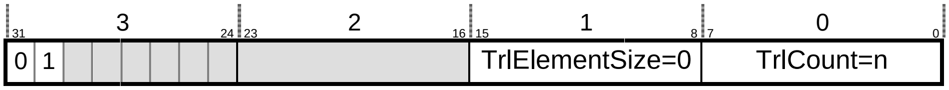


TRL Control Word

3
31
24
2
23
16
1
15
8
0
7
0
TransactionContext 1
TransactionDetailElement (mx4 bytes)
TransactionContext 2
TransactionDetailElement (mx4 bytes)
TransactionContext 3
TransactionDetailElement (mx4 bytes)
TransactionContext n
TransactionDetailElement (mx4 bytes)

Figure 3-32. Single Fixed Length Element

When TrlElementSize=0 in a single fixed-length TRL, the list simply contains TransactionContext values. Otherwise, each TransactionContext is followed by exactly one TransactionDetailElement that is exactly the length specified in TrlElementSize.



TRL Control Word

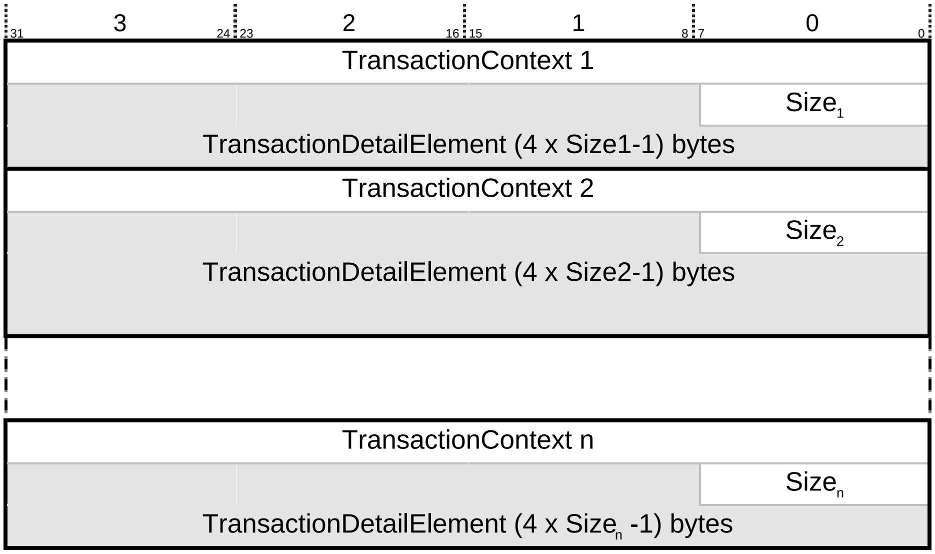
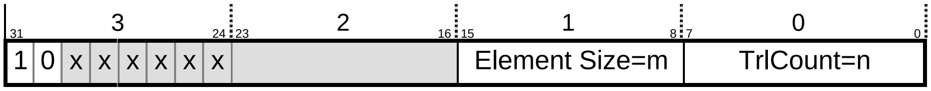

3
2
1
0
31
24
23
16
15
8
7
0
TransactionContext 1
Size_1
TransactionDetailElement (4 x Size1-1) bytes
TransactionContext 2
Size_2
TransactionDetailElement (4 x Size2-1) bytes
TransactionContext n
Size_n
TransactionDetailElement (4 x Size_n -1) bytes

Figure 3-33. Single Variable Length Element

For variable length TRLs, the size of the detail element (in 32-bit words) is indicated in the first byte of the transaction details. This value does not include the size of the TransactionContext, but does include the word that contains the size. Thus the value must not be zero. This allows a TrlElementSize from four to 1020 bytes.



TRL Control Word

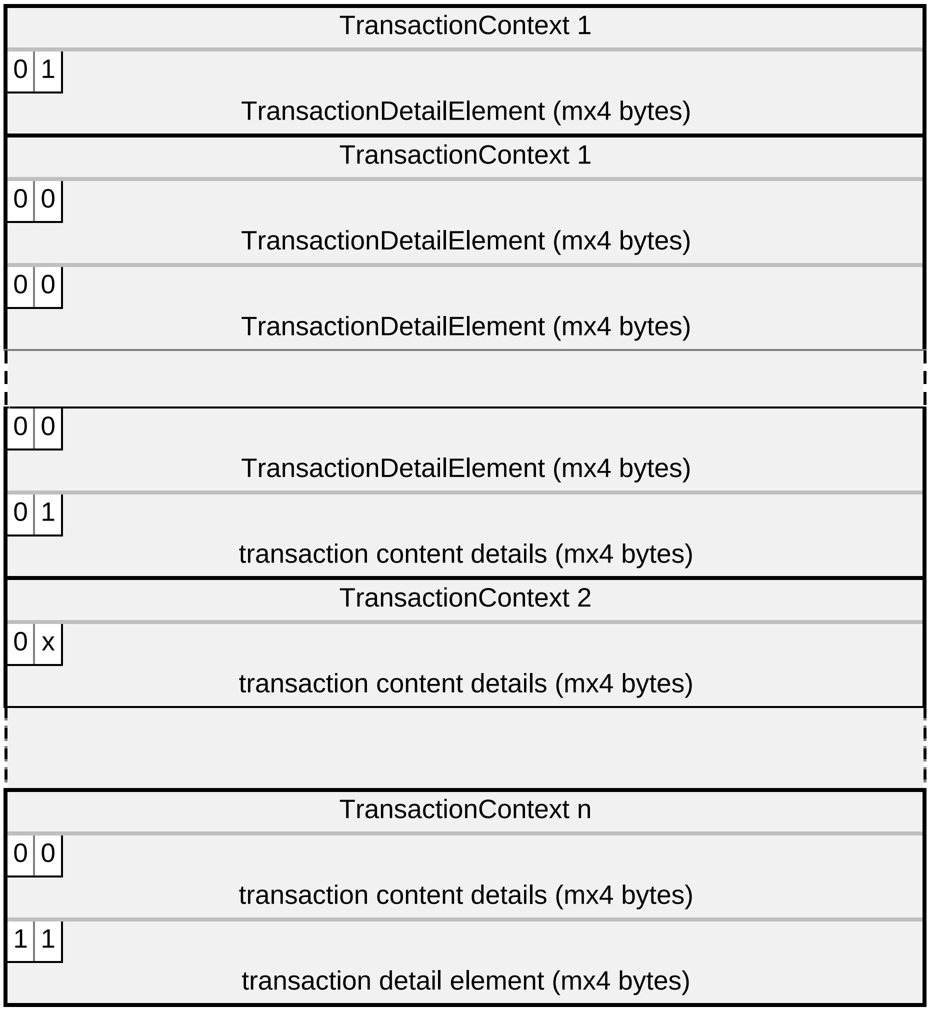

TransactionContext 1
0 1
TransactionDetailElement (mx4 bytes)
TransactionContext 1
0 0
TransactionDetailElement (mx4 bytes)
0 0
TransactionDetailElement (mx4 bytes)
0 0
TransactionDetailElement (mx4 bytes)
0 1
transaction content details (mx4 bytes)
TransactionContext 2
0 x
transaction content details (mx4 bytes)
TransactionContext n
0 0
transaction content details (mx4 bytes)
1 1
transaction detail element (mx4 bytes)

Figure 3-34. Multiple Fixed Length Elements

For a multiple fixed-length detail TRL, one or more fixed-length detail elements reside after each TransactionContext. Bits 30 and 31 of the first word of each detail element contain the flags that indicate whether the element is the last for the particular transaction context (bit 30) and whether this is the last element of the last context in the list (bit 31). The remainder of the element is reserved for class-specific definition. Each transaction element is the exact size specified in the TrlElementSize field, in the TrlControlWord.

3.4.4 Serial Numbers

For an I/O device to have redundant paths or be used in redundant systems, serial numbers are necessary to identify devices that represent the same entity. Each class defines a unique method for determining and presenting a device's serial number. The serial number reported for a device must be unique among devices of the same class and reported identically by all drivers that control that device. When a device such as a disk drive contains a removable medium, the serial number must be for the drive and not the medium.

The purpose of this requirement is not to ban devices with no serial number mechanisms, but rather to emphasize the need for such mechanisms. This specification does allow indicating that the serial number is unknown. The system integrator designing a system with redundant paths should verify that all devices residing on redundant paths report a valid serial number. The rapid growth of redundant paths and systems should encourage the hardware vendor to provide serial number capability.

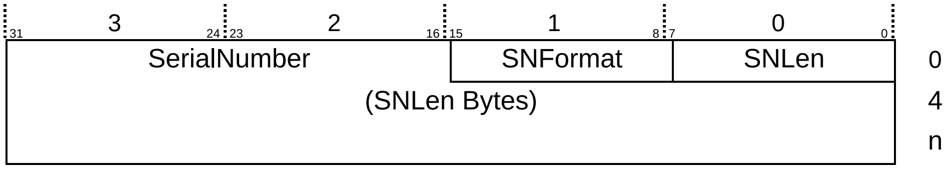


Figure 3-35. Format for Reporting Serial Number

The serial number is a variable-size field, whose first byte (SNLen) indicates the total bytes of the serial number that follows. For devices that do not report a serial number or if the serial number is unknown, the length field must be set to 0.

Immediately following the SNLen field is a SNFormat field. This one-byte field helps high-level management software determine how to display the serial number, as in the following table.

Table 3-8. Serial Number Formats

SNFormat	Description
0	Unknown
1	BINARY
2	ASCII
3	UNICODE (ISO/IEC -10646)
4	LAN MAC Address
5	WAN Access Address

When reporting LAN MAC Addresses as Serial Numbers, the following rules apply

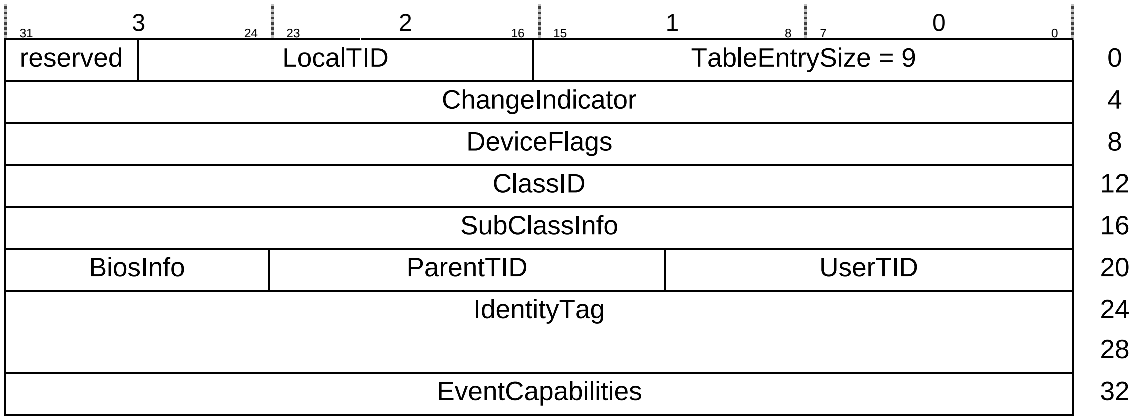
SN_Len — The serial number of a LAN adapter consists of a 48-bit universally-administered MAC address registered to the device. Therefore, this value is always 06h.

SN_Format — The serial number is a LAN MAC Address. Therefore, this value is always 04h.

Serial_Number — The serial number of a LAN adapter consists of a 48-bit universally administered MAC address registered to the device, specified in canonical format (i.e., the first bit of the address as transmitted on the wire is in the low-order bit of the first byte of this field). Note that this requires conversion of the address for big endian media such as Token Ring and FDDI.

3.4.5 Logical Configuration Table Entries

Each entry in an IOP's logical configuration table contains a SubClassInfo word. The structure of this word is defined by each class and identifies the major capabilities of the device. When the DDM registers a device, it provides the information for the LCT entry such as the device's ClassID and SubClassInfo. The IOP publishes this information in its logical configuration table, and the OSM uses this information when it determines which devices to query. See Chapter 4 for the structure of the table. LCT entries are defined in Figure 3-36.



31 3 24	23 2 16	15 1 8	7 0 0	
reserved	LocalTID	TableEntrySize = 9		0
ChangeIndicator				4
DeviceFlags				8
ClassID				12
SubClassInfo				16
BiosInfo	ParentTID	UserTID		20
IdentityTag				24
				28
EventCapabilities				32

Figure 3-36. LCT Entry Structure

Fields

BiosInfo

Identifier used to correlate a device with any connections the BIOS creates. This value is set by the BIOS via the ***ExecBiosInfoSet*** message. As an example, when a BIOS extension hooks INT13 for a storage device and is assigned drive ID 81h, this field is set to 81h to notify the OS not to call BIOS for drive ID 81h. The default value 0FFh indicates that the device is not the subject of a BIOS function call.

ChangeIndicator

Value of CurrentChangeIndicator last time this entry was updated.

ClassID

Message class of this device. Messages sent to the LocalTID must conform to this class definition. See Chapter 6 for the definition of ClassID structure.

DeviceFlags

Bit-specific field that identifies the device’s characteristics and capabilities.

Bit 0: Set to indicate that the device requests a configuration dialogue.

Bit 1: Set if the device can concurrently support more than one user.

Bit 4: Set if Peer Service Class is disabled (determined when the device is claimed by a primary service user). Peer service class is simply the ability of other DDMs to send base class messages to the device.

Bit 5: Set if the Management Service Class is disabled (determined when the device is claimed by a primary service user).

All other bits are reserved.

EventCapabilities

Each bit of this field corresponds to the same bit in the EventMask field of the ***UtilEventRegister*** message. A value of 1 indicates that the DDM can generate that type of event.

IdentityTag Part of serial number that uniquely identifies a device. This field is always eight bytes long and does not include the SNLen or SNFormat fields. If the serial number is fewer than 64 bits, it is pre-padded with zeros. If the serial number exceeds eight bytes, it is truncated to the lowest order eight bytes (those that provide unique identity). This field is used to match system configuration with the I/O device. If no serial number is known, then this field contains all zeros.

LocalTID Local target ID assigned by IOP to this device.

ParentTID TID of the device that created, registered, and manages this I/O device.

SubClassInfo This field is reserved for definition by each message class. Refer to the particular class section in Chapter 6. Unless specified otherwise, this field shall be the value returned in Parameter Group 0000h, Field 0, pre-padded with zeros.

TableEntrySize Number of 32-bit words consumed by this entry. This version of the specification defines 36 bytes, and thus a value of 9. Future versions of this specification may add fields to the end of this structure and, thus, will have values greater than nine.

UserTID TID of the primary service user of this device. Established by connection setup (***UtilClaim***), it indicates the OSM or ISM to which this resource is dedicated. The value of 0FFFh indicates that the resource is not allocated. A value other than FFFh indicates that the device is reserved.

3.4.6 Common Structures for Adapters

Both the shell and core specifications identify adapters by their bus type and location. The following definitions identify those attributes throughout this specification.

3.4.6.1 Bus Type

Table 3-9. Bus Type Code Assignments

Code point	Bus Type
00h	Local bus
01h	ISA bus
02h	EISA bus
03h	MCA bus
04h	PCI bus
05h	PCMCIA bus
06h	NuBus
07h	CardBus
80h	Other

3.4.6.2 Physical Location

PhysicalLocation — Eight bytes of data that identify the physical adapter or function. Format of this field depends on BusType, and those variations are defined in the following figures.

31 3 24	23 2 16	15 1 8	7 0 0	
reserved	PciBusNumber	PciDeviceNumber	PciFunctionNumber	0
PciDeviceID		PciVendorID		4

Figure 3-37. Structure of Physical Location for a PCI Bus Adapter

Fields

PciBusNumber — Identifies the bus by its assigned PCI identifier. PCI allows a maximum of 256 PCI buses within a single PCI system.

PciDeviceID — DeviceID field from PCI Configuration Header.

PciDeviceNumber — Identifies the specific PCI device on the bus. PCI allows a maximum of 32 PCI devices.

PciFunctionNumber — Identifies a particular function of the PCI device. PCI provides for 8 functions (0-7). This is a bit-specific field. Bit 0 corresponds to function 0 and bit 7 corresponds to function 7. The value 0FFh designates all functions of the PCI device. Note that each function of a PCI device meets the requirements of an adapter. This version of the specification requires that all functions within a particular device be unassigned, or assigned to the same IOP (but not necessarily assigned to the same DDM).

PciVendorID — VendorID field from PCI Configuration Header. A value of 0FFh indicates that no PCI device is present.

The PhysicalLocation format for Local bus, ISA bus, EISA bus, MCA bus, and Other Bus are shown below.

31 3 24	23 2 16	15 1 8	7 0 0	
reserved	reserved	BaseIOPort		0
BaseMemoryAddress				4

Figure 3-38. Structure of Physical Location for a Local Bus Adapter

31 3 24	23 2 16	15 1 8	7 0 0	
reserved	CSN	BaseIOPort		0
BaseMemoryAddress				4

Figure 3-39. Structure of Physical Location for an ISA Bus Adapter

31 3 24	23 2 16	15 1 8	7 0 0	
EisaSlotNumber	reserved	BaseIOPort		0
BaseMemoryAddress				4

Figure 3-40. Structure of Physical Location for an EISA Bus Adapter

31 3 24	23 2 16	15 1 8	7 0 0	
McaSlotNumber	reserved	BaseIOPort		0
BaseMemoryAddress				4

Figure 3-41. Structure of Physical Location for an MCA Bus Adapter

31 3 24	23 2 16	15 1 8	7 0 0	
reserved	reserved	BaseIOPort		0
BaseMemoryAddress				4

Figure 3-42. Structure of Physical Location for Other Bus

Fields

BaseIOPort — Identifies the specific adapter by its base I/O port address. A value of 0000h means not used, unknown, or otherwise not applicable.

BaseMemoryAddress — Identifies the specific adapter by its base memory address. A value of 0000-0000h means not used, unknown, or otherwise not applicable.

EisaSlotNumber — Identifies the adapter by its EISA card slot. A value of 0FFh means not used, unknown, or otherwise not applicable.

McaSlotNumber — Identifies the adapter by its MCA card slot. A value of 0FFh means not used, unknown, or otherwise not applicable.

PCMCIA adapter: PhysicalLocation format is not defined.

NuBus adapter: PhysicalLocation format is not defined.

CardBus adapter: PhysicalLocation format is not defined.

3.4.7 Managing I$_2$O Devices

3.4.7.1 Device Management Model

The I$_2$O administration facility provides mechanisms for accessing collections of managed objects called *parameter groups*. A parameter group may contain configuration parameters, statistical information, control variables, and so on. Objects in groups may be read-write or read-only.

Each parameter group (identified by a GroupNumber) contains columns of data objects known as *fields* and, optionally, multiple rows of some replication of those fields. There are two distinct types of groups: *scalar* groups, containing only a single row, and *table* groups, containing multiple rows. In a table group, rows are selected by the value of the *key field*, which is defined as the first field of the group. Thus for a table group, a [GroupNumber, FieldIdx, KeyValue] triple identifies a single parameter. For a scalar group, a [GroupNumber, FieldIdx] pair identifies a single parameter.

GroupNumbers are 16-bit integer values allocated by category with ranges allocated to generic groups (which pertain to all devices), class groups (which pertain to all devices of a particular class), and private groups (available for vendor- or driver-specific assignment). The following table identifies categories of parameters by ranges of their group IDs.

Table 3-10: Parameter GroupNumber Ranges

GroupNumber			Category
0000	to	0FFF	Class-Specific Parameter Groups
1000	to	7FFF	reserved
8000	to	8FFF	Private Parameter Groups
9000	to	EFFF	reserved
F000	to	FFFF	Generic Parameter Groups

A FieldIdx is a 16-bit integer derived from the sequential index of the field's position in the row, starting with 0. Negative values for FieldIdx have special meaning, thus a parameter group may contain up to 7FFFh (32767) fields. The maximum size of a field is 255 bytes.

For table groups, the first field (FieldIdx=0) is the *key* field used to select or specify a particular row. This value must uniquely identify the row in a way that does not change. For example, the key field may not be the row index if rows can be added or deleted from the table (except for additions to its end). Such a KeyValue for a particular row would change arbitrarily. A table group may contain up to FFFFh (65535) rows.

FieldIdx = 0	**FieldIdx = 1**	**FieldIdx = 2**		**FieldIdx = n**
Parameter$_0$	Parameter$_1$	Parameter$_2$	…	Parameter$_n$

Figure 3-43: Model of a Scalar Group

(Key field)				
FieldIdx = 0	**FieldIdx = 1**	**FieldIdx = 2**		**FieldIdx = n**
KeyValue$_0$	Parameter$_1$(KeyValue$_0$)	Parameter$_2$(KeyValue$_0$)	…	Parameter$_n$(KeyValue$_0$)
KeyValue$_1$	Parameter$_1$(KeyValue$_1$)	Parameter$_2$(KeyValue$_1$)	…	Parameter$_n$(KeyValue$_1$)
KeyValue$_2$	Parameter$_1$(KeyValue$_2$)	Parameter$_2$(KeyValue$_2$)	…	Parameter$_n$(KeyValue$_2$)
: :	: :	: :	: :	: :
KeyValue$_m$	Parameter$_1$(KeyValue$_m$)	Parameter$_2$(KeyValue$_m$)	…	Parameter$_n$(KeyValue$_m$)

Each row provides an identical set of parameters for a different KeyValue, with each parameter's value depending on the KeyValue.

Figure 3-44: Model of a Table Group

3.4.7.2 Basic Parameter Group Access

Chapter 6 provides the following utility messages for accessing the managed objects of a device by its defined parameter groups:

Table 3-11: Parameter Access Messages

Message Name	Description
UtilParamsGet	Get field sizes, read parameters or rows of parameters.
UtilParamsSet	Modify parameters, add or delete rows.

Each message supports a number of operations on a device's parameter tables. The following sections describe in detail those operations. These messages actually specify an *operations list*. The list describes a sequence of operations, each of which is specific to a particular parameter group. Each operation is described in an OperationBlock.

The device parses through the Operations List performing the operations in the order listed. It creates a *Results List* that is copied to the *Result Buffer*. Under normal operation, one ResultBlock is listed in the Results List for each operation in the Operations List; the ResultBlocks appear in the same order as their corresponding OperationBlocks. The only exception to this is when the Result Buffer provided is too small for each operation to report a result (see section 3.4.7.5, *Error Reporting*).

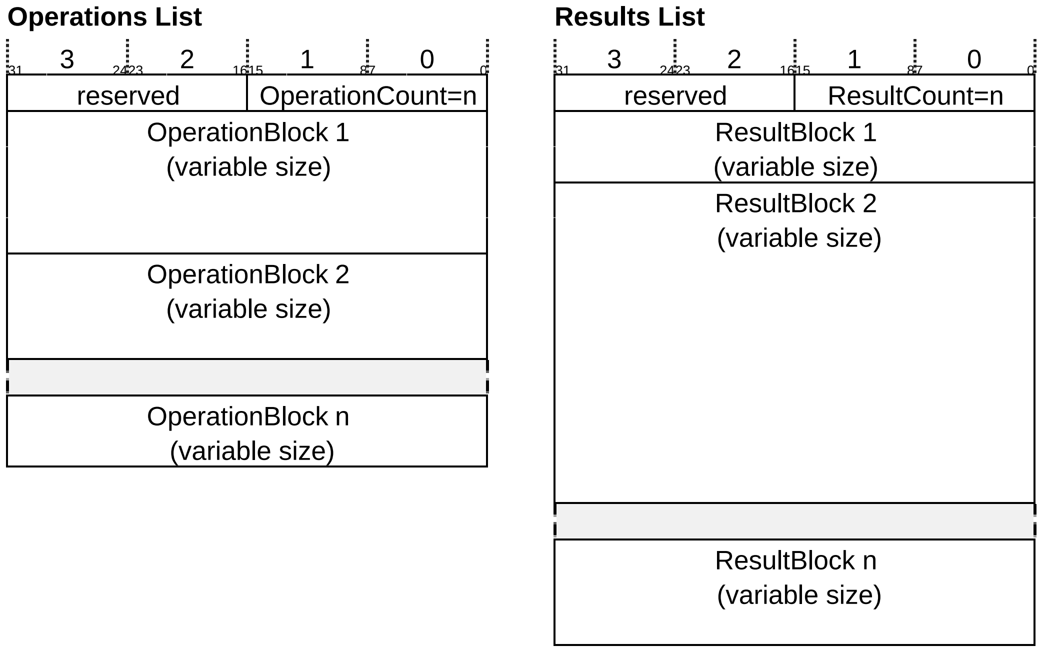


Figure 3-45: Operations List And Results List Structures

The size of each OperationBlock depends on the type of operation, as well as the number and size of the fields it involves. The size of each ResultBlock depends on the type of operation, its status, and the number and size of the fields it involves. Each OperationBlock and ResultBlock is padded to end on a four-byte boundary.

3.4.7.3 Reading Device Parameters

The request consists of one or more query operations described in Table 3-12.

Table 3-12: Parameter Read Operations

Operation	Operands/ Arguments	ResultBlock	Description
FIELD_GET	GroupNumber FieldCount ListOfFieldIdx*	BlockSize, BlockStatus ListOfValues* ErrorInfo	Returns values of selected fields in a scalar group
LIST_GET	GroupNumber FieldCount ListOfFieldIdx* KeyCount ListOfKeyValues*	BlockSize, BlockStatus ListOfValues* ErrorInfo	Returns values of selected fields of selected rows in a table group
MORE_GET	GroupNumber FieldCount ListOfFieldIdx* PreviousKey	BlockSize, BlockStatus RowCount moreFlag ListOfValues* ErrorInfo	Returns values of selected fields of all rows following a specified row in a table group
SIZE_GET	GroupNumber FieldCount ListOfFieldIdx*	BlockSize, BlockStatus ListOfFieldSizes* ErrorInfo	Returns size in bytes of selected fields
TABLE_GET	GroupNumber FieldCount ListOfFieldIdx*	BlockSize, BlockStatus RowCount moreFlag ListOfValues* ErrorInfo	Returns values of selected fields of all rows in a table group

3.4.7.3.1 Features common to all READ operations

All read operations take as a minimum a GroupNumber and a field list as arguments. The GroupNumber identifies the parameter group on which the operation is performed. The field list is a FieldCount followed by a list of FieldIdx values. It indicates which fields in the group the target returns in the results buffer.

The FieldCount normally specifies how many 16-bit FieldIdx values follow, but a FieldCount of -1 (FFFFh) returns all fields in the group. In this case, no FieldIdx values are specified after the FieldCount.

The FieldIdx arguments normally specify the index of fields and thus are in the range of 0000h to 7FFFh. A negative value (FFFFh to 8000h) has special meaning; negative values other than 8000h cause byte padding to be inserted into the result at a specified point. This feature helps ensure that data objects in the result are correctly aligned, for easy parsing. For example, a

FieldIdx value of -3 (FFFDh) causes three bytes of padding to be inserted into the result. This is useful for aligning a 32-bit field after reading an eight-bit field.

A FieldIdx value of 8000h is a placeholder in the request, and it places no bytes into the result. This value is otherwise ignored by the target. This value offers limited use with read operations, but is useful in write operations.

All operations return a ResultBlock, containing a BlockStatus field. Under normal operation, this value is zero and the Error Information field is null. If an error occurs during an operation, error information is appended to the ResultBlock and the ErrorInfoSize is set to reflect its size. Zero or more bytes of padding may appear at the end of the Error Information fields. Section 3.4.7.5, *Error Reporting*, provides more information on error codes and the Error Information field.

All read operations are executed by: Operations on multiple rows (such as *LIST_GET*) first complete all actions on the first row specified, then the actions on the next row, and so on. This becomes significant when an error occurs, to determine how far an operation progressed before the error.

Some operations on tables allow selecting the rows. For these cases, a key list is supplied. The key list is a KeyCount followed by KeyValues. Each KeyValue identifies the rows.

Each operation returns a ResultBlock, as shown in Figure 3-46.

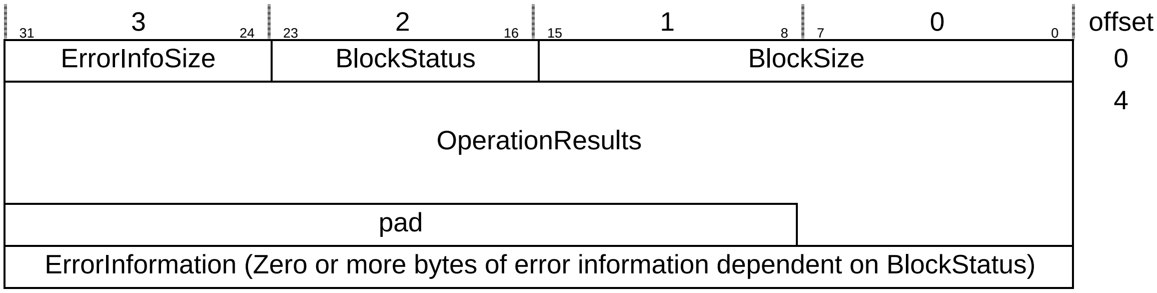

31
3
24
23
2
16
15
1
8
7
0
0
offset
ErrorInfoSize
BlockStatus
BlockSize
0
4
OperationResults
pad
ErrorInformation (Zero or more bytes of error information dependent on BlockStatus)

Figure 3-46. ResultBlock Template for Read Operations

Fields

BlockSize — Specifies the size of this ResultBlock in number of 32-bit words. This includes BlockSize, BlockStatus, Operation Results, Pad, and Error Information fields. The value zero means the size exceeds the 256K byte maximum limit of the BlockSize value.

BlockStatus — Indicates the status of the operation as specified in Table 3-14.

ErrorInformation — Contains zero or more bytes of error information on the operation, as in Figure 3-47. Depending on the error, some fields of the Error Information Template are not included in the Error Information field.

ErrorInfoSize — Specifies the size of the Error Information field in number of 32-bit words. A zero means there is no error information.

OperationResults — Contains the results of the operation.

Pad — 0, 1, 2, or 3 bytes of padding that make the block end on a 32-bit boundary. The value of pad bytes is undefined.

31 3 24	23 2 16	15 1 8	7 0 0	offset
GroupNumber		OperationCode		0
NumberKeys	AdditionalStatus	FieldIdx		2
ListOfKeyValues (variable size, variable count)				
pad (variable size)				

Figure 3-47. Error Information Template

Fields

AdditionalStatus	reserved
FieldIdx	FieldIdx of the field generating the error.
GroupNumber	GroupNumber copied from the Operations List for this operation.
ListOfKeyValues	List of key values that generated this error. The subset of the KeyValues in the Operations List
NumberKeys	Number of KeyValues listed in this error report. A value of zero means that no KeyValues are specified.
OperationCode	Operation code for this operation, copied from the Operations List.
Pad	Zero or more bytes that make the Error Information field end on a 32-bit boundary. The value of a pad byte is undefined.

3.4.7.3.2 *FIELD_GET* Operation

The *FIELD_GET* operation returns the values of selected fields in a scalar group. It is an error to invoke this operation on a table group. The structure of the *FIELD_GET* operation is shown below:

For Specific Fields

16 1 8	7 0 0	offset
Operation = FIELD_GET		0
GroupNumber		2
FieldCount = n		4
FieldIdx 1		6
FieldIdx 2		8
FieldIdx n		4+2n
Pad (if necessary)		

For All Fields

16 1 8	7 0 0	offset
Operation = FIELD_GET		0
GroupNumber		2
FieldCount = -1		4
Pad		6

Figure 3-48: *FIELD_GET* Operation Block

The GET VALUE operation consists of a GroupNumber followed by a field list. The field list contains a FieldCount and zero or more FieldIdx values.

Field values in the result are returned sequentially in the same order as their indices appeared in the OperationBlock. If the Result Buffer is too small to hold the output from a *FIELD_GET* operation, a *PARAMS_STATUS_BUFFER_FULL* error is returned. For a full description of how errors are reported, see section 3.4.7.5, *Error Reporting*.

31 3 24	23 2 16	15 1 8	7 0 0	offset
ErrorInfoSize	BlockStatus	BlockSize		0
ListOfValues List for field values in the order specified in the request block				4
pad (as necessary)				
ErrorInformation (Zero or more bytes of error information dependent on BlockStatus)				

Figure 3-49. *FIELD_GET* Operation ResultBlock

3.4.7.3.3 *LIST_GET* Operation

The *LIST_GET* operation returns the values of selected fields from selected rows in a table group. It is an error to invoke this operation on a scalar group. The structure of the *LIST_GET* OperationBlock is shown in the following figure:

For Specific Fields

16 1 8	7 0 0	offset
Operation = LIST_GET		0
GroupNumber		2
FieldCount = n		4
FieldIdx 1		6
FieldIdx 2		8
FieldIdx n		4+2n
KeyCount = m		4+2n+2
KeyValue 1 (size dependent on Key field)		
KeyValue 2 (size dependent on Key field)		
KeyValue m (size dependent on Key field)		
Pad (as necessary)		

For All Fields

16 1 8	7 0 0	offset
Operation = LIST_GET		0
GroupNumber		2
FieldCount = -1		4
KeyCount = m		6
KeyValue 1 (size dependent on Key field)		8
KeyValue 2 (size dependent on Key field)		
KeyValue m (size dependent on Key field)		
Pad (as necessary)		

Figure 3-50. *LIST_GET* Operation Block

KeyCount = number of KeyValue fields that follow.
Note that the size of the KeyValue field varies by group.

The key values passed in the OperationBlock specifies the rows for which the target returns the specified field values. Field values are listed in row-major order: The specified field values for the first specified row are listed first, followed by field values for the next row, and so on.

If the result buffer is too small to hold the output from a *LIST_GET* operation, a *PARAMS_STATUS_BUFFER_FULL* error is returned. For a full description of how errors are reported, see section 3.4.7.5, *Error Reporting*.

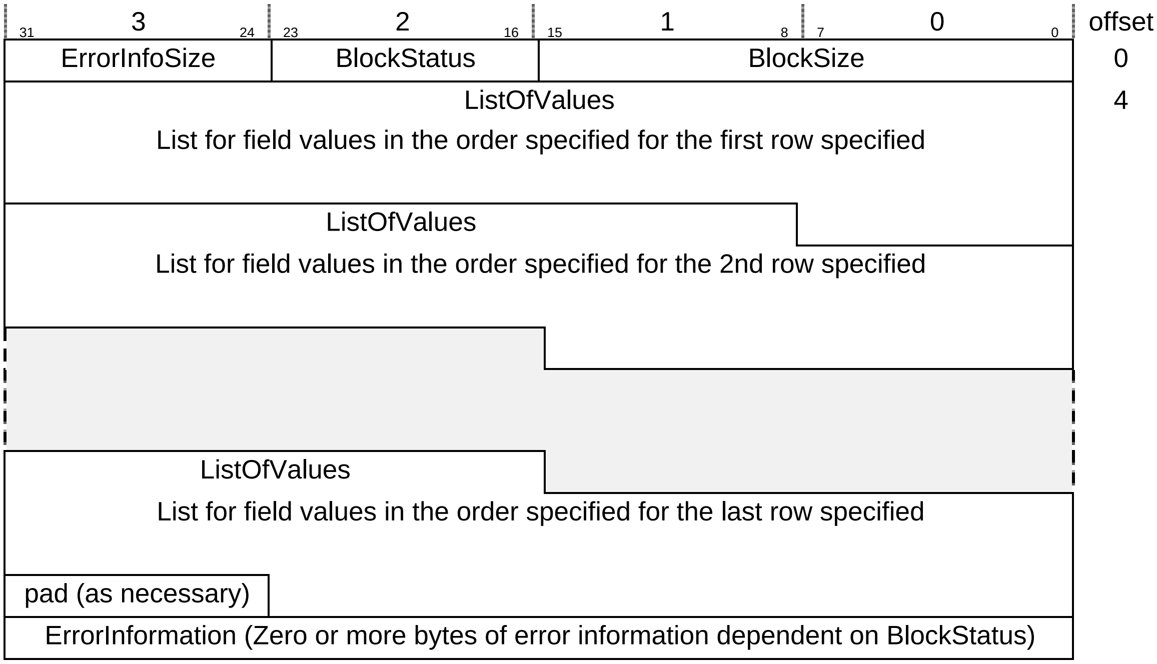

31
3
24
23
2
16
15
1
8
7
0
0
offset
ErrorInfoSize
BlockStatus
BlockSize
0
ListOfValues
4
List for field values in the order specified for the first row specified
ListOfValues
List for field values in the order specified for the 2nd row specified
ListOfValues
List for field values in the order specified for the last row specified
pad (as necessary)
ErrorInformation (Zero or more bytes of error information dependent on BlockStatus)

Figure 3-51. *LIST_GET* Operation ResultBlock

3.4.7.3.4 *MORE_GET* Operation

The *MORE_GET* operation returns the values of selected fields from all rows in a table group, starting immediately after the row specified by the PreviousKey field. Use this command following a *TABLE_GET* operation to fetch rows from a table that is too large to read in a single operation.

It is an error to invoke this operation on a scalar group. The OperationBlock for the *MORE_GET* operation is shown below:

For Specific Fields

16 1 8	7 0 0	offset
Operation = MORE_GET		0
GroupNumber		2
FieldCount = n		4
FieldIdx 1		6
FieldIdx 2		8
FieldIdx n		4+2n
PreviousKey (size varies by Key field)		6+2n
Pad		

For All Fields

16 1 8	7 0 0	offset
Operation = MORE_GET		0
GroupNumber		2
FieldCount = -1		4
PreviousKey (size varies by Key field)		
Pad		

Figure 3-52. ***MORE_GET*** **Operation Block**

Note that the size of the PreviousKey field varies depending on the group.

The structure of the ResultBlock is the same as for the *TABLE_GET* operation in Figure 3-56. The RowCount value indicates how many complete rows of field values were returned. When all remaining rows of the table are returned, the MoreFlag is set to zero. Otherwise, it is set to a non-zero value. The remaining rows of the table may be fetched using additional *MORE_GET* operations.

Field values are listed in row-major order. The specified field values for the first row are first, followed by field values for the second row, and so on.

For a full description of how errors are reported, see section 3.4.7.5, *Error Reporting*.

3.4.7.3.5 *SIZE_GET* Operation

The *SIZE_GET* operation returns the size in bytes of selected fields in a group. This permits discovery field sizes at run-time by generic clients without a priori knowledge. However, in general, field sizes are defined as part of this specification and will be known a priori by specific clients. The structure for the *SIZE_GET* OperationBlock is shown below:

For Specific Fields

16 1 8	7 0 0	offset
Operation = SIZE_GET		0
GroupNumber		2
FieldCount = n		4
FieldIdx 1		6
FieldIdx 2		8
FieldIdx n		4+2n
Pad (if necessary)		4+2n

For All Fields

16 1 8	7 0 0	offset
Operation = SIZE_GET		0
GroupNumber		2
FieldCount = -1		4
Pad		6

Figure 3-53. ***SIZE_GET*** **Operation Block**

Processing the *SIZE_GET* operation lists the field size values in the result sequentially, in the same order as their FieldIdx values appeared in the OperationBlock. If the result buffer is too small to hold the output from a *SIZE_GET* operation, a *PARAMS_STATUS_BUFFER_FULL* error is returned. For a full description of how errors are reported, see section 3.4.7.5, *Error Reporting*.

31 3 24	23 2 16	15 1 8	7 0 0	offset
ErrorInfoSize	BlockStatus	BlockSize		0
Field4 Size	Field 3 Size	Field 2 Size	Field 1 Size	4
pad	Field n-1 Size	Field n-2 Size	Field n-3 Size	
ErrorInformation (zero or more bytes of error information, depending on BlockStatus)				

Figure 3-54. *SIZE_GET* Operation ResultBlock

3.4.7.3.6 *TABLE_GET* Operation

The *TABLE_GET* operation returns the values of selected fields from all rows in a table group. It is an error to invoke this operation on a scalar group. The structure for the *TABLE_GET* OperationBlock is shown below:

For Specific Fields

16 1 8	7 0 0	offset
Operation = TABLE_GET		0
GroupNumber		2
FieldCount = n		4
FieldIdx 1		6
FieldIdx 2		8
FieldIdx n		4+2n
Pad (if necessary)		

For All Fields

16 1 8	7 0 0	offset
Operation = TABLE_GET		0
GroupNumber		2
FieldCount = -1		4
Pad		6

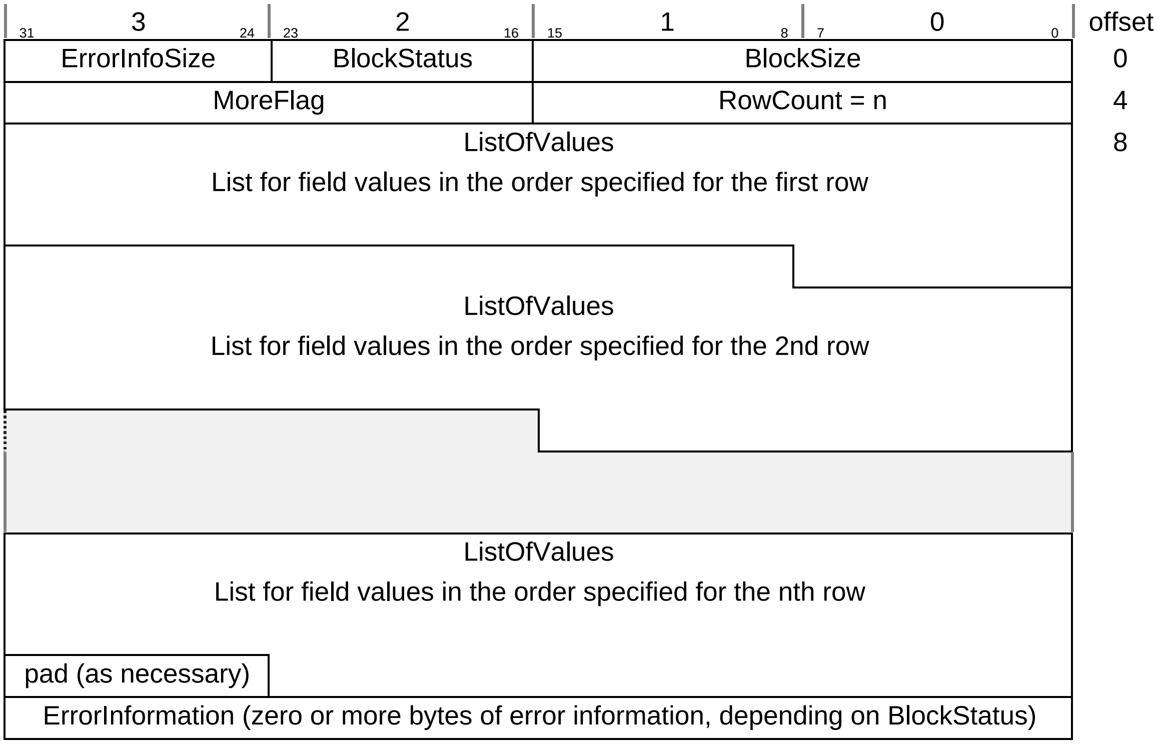
Figure 3-55. *TABLE_GET* Operation Block

The *TABLE_GET* operation consists of a group number, followed by a field list.

The RowCount value in the result indicates the exact number of complete rows returned. If all rows of the table cannot be returned in the result buffer, the MoreFlag is set to a non-zero value (an error is not returned). The remaining rows of the table can be fetched using one or more *MORE_GET* operations. If the result contains all rows, the MoreFlag is set to zero.

Field values are listed in row-major order: the specified field values for the first row, followed by field values for the second row, and so on.

For a full description of how errors are reported, see section 3.4.7.5, *Error Reporting*.



31 3 24	23 2 16	15 1 8	7 0 0	offset
ErrorInfoSize	BlockStatus	BlockSize		0
MoreFlag		RowCount = n		4
ListOfValues List for field values in the order specified for the first row				8
ListOfValues List for field values in the order specified for the 2nd row				
ListOfValues List for field values in the order specified for the nth row				
pad (as necessary)				
ErrorInformation (zero or more bytes of error information, depending on BlockStatus)				

Figure 3-56. *TABLE_GET* Operation ResultBlock

3.4.7.4 Modifying Device Parameters

The Operations List for an ***UtilParamSet*** request consists of one or more operations described in Table 3-13.

Table 3-13: Parameter Modify Operations

Operation	Operands/ Arguments	ResultBlock	Description
FIELD_SET	GroupNumber FieldCount ListOfFieldIdx* ListOfValues*	BlockSize, BlockStatus ErrorInfo	Sets values of selected fields in a scalar group
LIST_SET	GroupNumber FieldCount ListOfFieldIdx* RowCount ListOfValues*	BlockSize, BlockStatus ErrorInfo	Sets values of selected fields of selected rows in a table group
ROW_ADD	GroupNumber FieldCount ListOfFieldIdx* RowCount ListOfValues*	BlockSize, BlockStatus ErrorInfo	Adds rows to a table group
ROW_DELETE	GroupNumber KeyCount ListOfKeyValues*	BlockSize, BlockStatus ErrorInfo	Deletes rows from a table group
TABLE_CLEAR	GroupNumber	BlockSize, BlockStatus ErrorInfo	Deletes all rows from a table group

3.4.7.4.1 Features common to all MODIFY operations

All modify operations are executed in a row-oriented fashion. This means that operations acting over multiple rows (such as *LIST_SET*) complete all actions on the first row specified, then the actions on the next row, and so on. This becomes significant if an error occurs, to determine how far an operation progressed before the error.

The FieldIdx arguments normally specify the index of fields, and thus are in the range of 0000h to 7FFFh. A negative value (FFFFh to 8000h) has special meaning; those other than 8000h identify byte padding in the value list. Use this feature to provide aligned data objects for creating lists easily. For example, a field index of -3 (FFFDh) tells the target to skip three bytes before reading the next field value. Use this for aligning a 32-bit field, after specifying an eight-bit field.

A FieldIdx value of 8000h is a placeholder in the request, and requires no corresponding value in the values list. This FieldIdx value is otherwise ignored by the target. These two features allow the value list to always align on a 32-bit boundary, and specific values to align on a desired boundary. This simplifies building the value list using high-level language constructs.

Some operations on tables allow selecting the rows. For these cases, a key list is supplied. It contains a KeyCount, followed by a list of KeyValues, each of which identifies the row.

All operations return a ResultBlock, containing a BlockStatus field, as in Figure 3-57. (See Figure 3-46 for field definitions.) Under normal operation, this value is zero and the ErrorInformation field is null. If an error occurs during an operation, ErrorInformation is appended to the ResultBlock and the ErrorInfoSize is set to reflect its size. Zero or more bytes of padding may appear at the end of the ErrorInformation fields. Section 3.4.7.5, *Error Reporting*, provides more information on error codes and the ErrorInformation field.

31 3 24	23 2 16	15 1 8	7 0 0	offset
ErrorInfoSize	BlockStatus	BlockSize		0
ErrorInformation (zero or more bytes of error information, depending on BlockStatus)				

Figure 3-57. ResultBlock Template for MODIFY Operations

If an error occurs when processing a modify operation, all parts of the operation up to the point of failure are assumed complete. Any parts of the operation following the failure are not performed, unless otherwise stated.

The requester determines the point of failure by the error information in the ResultBlock. See section 3.4.7.5, *Error Reporting*.

3.4.7.4.2 *FIELD_SET* Operation

The *FIELD_SET* operation modifies the values of selected fields in a scalar group. It is an error to invoke this operation on a table group. The structure for the *FIELD_SET* operation is shown in the following figure:

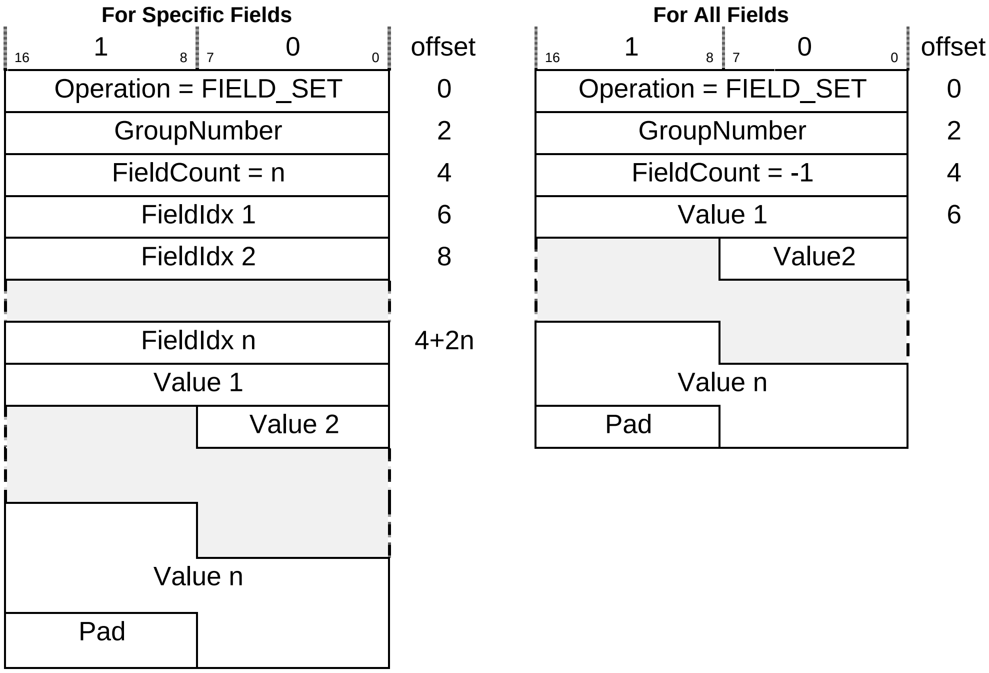


Figure 3-58: *FIELD_SET* Operation Block

The *FIELD_SET* operation consists of a group number followed by a field list. The field list contains a FieldCount plus zero or more FieldIdx values. Following the field list are the values for those fields, in the same order as the field list.

The FieldCount argument may be specified as -1 (FFFFh) indicating that all fields in the group are to be written. For this case, no field indices are specified.

If a group contains fields that are read-only, it is impossible to write the entire group using a wildcard field count. In this instance, the *FIELD_SET* operation halts with a *PARAMS_STATUS_FIELD_UNWRITEABLE* error when it reaches the first read-only field, and fields beyond that point are not modified. See section 3.4.7.5, *Error Reporting*.

31 3 24	23 2 16	15 1 8	7 0 0	offset
ErrorInfoSize	BlockStatus	BlockSize		0
ErrorInformation (zero or more bytes of error information, depending on BlockStatus)				

Figure 3-59. SET VALUE Operation ResultBlock

3.4.7.4.3 *LIST_SET* Operation

The *LIST_SET* operation sets the values of selected fields in each specified row of a table group. It is an error to invoke this operation on a scalar group. The structure of the *LIST_SET* OperationBlock is shown in the following figure:

For Specific Fields

16 1 8	7 0 0	offset
Operation = LIST_SET		0
GroupNumber		2
FieldCount = n		4
FieldIdx 1 = 0		6
FieldIdx 2		8
FieldIdx n		4+2n
RowCount = m		4+2n+2
KeyValue 1 †		
value (FieldIdx 2 , KeyValue 1) †		
value (FieldIdx 3 , KeyValue 1) †		
value (FieldIdx n , KeyValue 1) †		
KeyValue 2 †		
value (FieldIdx 2 , KeyValue 2) †		
value (FieldIdx 3 , KeyValue 2) †		
value (FieldIdx n , KeyValue 2) †		
KeyValue m †		
value (FieldIdx 2 , KeyValue m) †		
value (FieldIdx 3 , KeyValue m) †		
value (FieldIdx n , KeyValue m) †		

† field sizes vary, last value padded as necessary

For All Fields

16 1 8	7 0 0	offset
Operation = LIST_SET		0
GroupNumber		2
FieldCount = -1		4
RowCount = m		6
KeyValue1 †		8
value (FieldIdx 2 , KeyValue1) †		
value (FieldIdx 3 , KeyValue1) †		
value (FieldIdx n , KeyValue 1) †		
KeyValue 2 †		
value (FieldIdx 2 , KeyValue 2) †		
value (FieldIdx 3 , KeyValue 2) †		
value (FieldIdx n , KeyValue 2) †		
KeyValuem †		
value (FieldIdx 2 , KeyValue m) †		
value (FieldIdx 2 , KeyValue m) †		
value (FieldIdx n , KeyValue m) †		

† field sizes vary, last value padded as necessary

Figure 3-60. LIST_SET Operation Block

The *LIST_SET* operation consists of a group number, a field list (FieldCount followed by zero or more field indices), and a value list. The value list is in row-major order: All field values for the first row are listed, followed by all field values for the second, and so on.

The first entry in the field index list must be the key field (FieldIdx=0), unless a wildcard FieldCount of -1 (FFFFh) is specified for FieldCount: That indicates that the values for all fields of the row are provided. In either case, the first field in the value list for each row must be the key field value.

Negative FieldIdx values indicate padding bytes in the list of parameter values. Thus, a value of -n means to skip the next n bytes in the list. A FieldIdx value of 8000h is a placeholder and means to do nothing.

If a group contains read-only fields, it is impossible to write the entire group using a wildcard field count. In this instance, the *LIST_SET* operation halts with a *PARAMS_STATUS_FIELD_UNWRITEABLE* error when it reaches the read-only field, and fields beyond that are not modified.

By nature of the operation, the key field can never be modified. To effectively change the key field value, a new row must be added and the old row deleted.

3.4.7.4.4 *ROW_ADD* Operation

The *ROW_ADD* operation adds rows to a table group. It is an error to invoke this operation on a scalar group or a table group that does not support dynamic addition of rows. The structure of the *ROW_ADD* OperationBlock is shown in the following figure:

For Specific Fields

16 1 8 ⁞ 7 0 0	offset
Operation = *ROW_ADD*	0
GroupNumber	2
FieldCount = n	4
FieldIdx$_1$ = 0	6
FieldIdx$_2$	8
FieldIdx$_n$	4+2n
RowCount = m	4+2n+2
KeyValue$_1$ †	
value (FieldIdx$_2$, KeyValue$_1$) †	
value (FieldIdx$_3$, KeyValue$_1$) †	
value (FieldIdx$_n$, KeyValue$_1$) †	
KeyValue$_2$ †	
value (FieldIdx$_2$, KeyValue$_2$) †	
value (FieldIdx$_3$, KeyValue$_2$) †	
value (FieldIdx$_n$, KeyValue$_2$) †	
KeyValue$_m$ †	
value (FieldIdx$_2$, KeyValue$_m$) †	
value (FieldIdx$_3$, KeyValue$_m$) †	
value (FieldIdx$_n$, KeyValue$_m$) †	

† field sizes vary, last value padded as necessary

For All Fields

16 1 8 ⁞ 7 0 0	offset
Operation = *ROW_ADD*	0
GroupNumber	2
FieldCount = -1	4
RowCount = m	6
KeyValue$_1$ †	8
value (FieldIdx$_2$, KeyValue$_1$) †	
value (FieldIdx$_3$, KeyValue$_1$) †	
value (FieldIdx$_n$, KeyValue$_1$) †	
KeyValue$_2$ †	
value (FieldIdx$_2$, KeyValue$_2$) †	
value (FieldIdx$_3$, KeyValue$_2$) †	
value (FieldIdx$_n$, KeyValue$_2$) †	
KeyValue$_m$ †	
value (FieldIdx$_2$, KeyValue$_m$) †	
value (FieldIdx$_2$, KeyValue$_m$) †	
value (FieldIdx$_n$, KeyValue$_m$) †	

† field sizes vary, last value padded as necessary

Figure 3-61. *ROW_ADD* Operation Block

The *ROW_ADD* operation consists of a group number, a field list (FieldCount followed by zero or more field indices), and a value list. The value list is in row-major order: All field values for the first row are listed, followed by all field values for the second, and so on.

The first entry in the field index list must be the key field (FieldIdx=0), unless a wildcard FieldCount of -1 (FFFFh) is specified for FieldCount: That indicates that the values for all fields of the row are provided. In either case, the first field in the value list for each row must be the key field value.

Negative FieldIdx values indicate padding bytes in the list of parameter values. Thus, a value of -n means to skip the next n bytes in the list. A FieldIdx value of 8000h is a placeholder and means to do nothing.

The only difference between the *LIST_SET* and *ROW_ADD* operations is that *LIST_SET* requires the key value already in the table and *ROW_ADD* requires that it not exist. Otherwise, their formats and operations are identical.

It is not necessary to provide values for all fields in a row when adding rows to a table. The operation must specify the key field value as a minimum: The DDM should provide sensible default values for fields whose values are absent from the value list.

3.4.7.4.5 *ROW_DELETE* Operation

The *ROW_DELETE* operation deletes rows from a table group. It is an error to invoke this operation on a scalar group or on a table group that does not support dynamic deletion of rows. The structure of the *ROW_DELETE* OperationBlock is shown below:

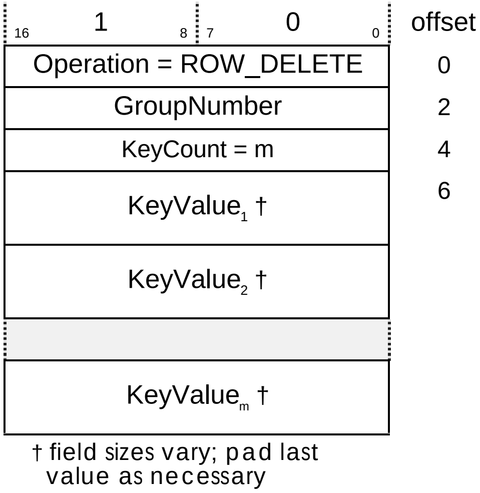


Figure 3-62. *ROW_DELETE* Operation Block

ROW_DELETE consists of a group number and a key list (KeyCount followed by zero or more key values). Note that the sizes of the KeyValue arguments vary, depending on the size of the key for the group.

3.4.7.4.6 *TABLE_CLEAR* Operation

The *TABLE_CLEAR* operation is used to delete all rows from a table group. It is an error to invoke this operation on a scalar group or a table group that does not support *TABLE_CLEAR*. A sample OperationBlock for *TABLE_CLEAR* is shown below:

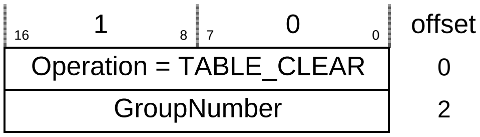


Figure 3-63. *TABLE_CLEAR* Operation Block

TABLE_CLEAR consists simply of a number specifying the group to clear.

3.4.7.5 Error Reporting

3.4.7.5.1 Errors reported in the reply frame header

If all operations complete without error, the reply returns a successful status. If the ResultsList did not copy to the Results buffer, the reply indicates *ERROR_NO_DATA_TRANSFER.* If any operation encounters an error and any portion of an OperationBlock completes, then the reply indicates ERROR_PARTIAL_TRANSFER. In this case, the error status in the appropriate ResultsBlock indicates the nature of the error.

3.4.7.5.2 Errors Reported in Result Blocks

Errors during parsing the operations return information to the ResultBlock for the failed operation. The module processing an OperationBlock halts immediately when it detects an error, and appends error information to the ResultBlock.

The length of error information varies and has zero or more bytes of padding to maintain four-byte alignment. This guarantees alignment of the following ResultBlocks.

3.4.7.5.3 Determining the Point of Failure within an Operation

The error information returned is minimal but always sufficient to determine the point of failure within the operation. Certain *operation level* errors are detected before any parameter accesses occur: They return an ErrorCode, the OperationCode, and the GroupNumber. No part of the operation is performed.

Other errors are detected as the operation is undertaken. Most errors return a field index as an additional item in the error report. For *field level* errors, this is sufficient to identify the point of failure, even if the operation acted over multiple rows. The error will have occurred on the first row processed, since all multi-row operations act on one row at a time.

Some errors are specific to certain rows. These *row level* errors return a KeyValue, identifying the row where the error occurred. No operations on that row will have been undertaken.

This leaves errors that occur at a specific data element, i.e. a specific field in a specific row. These *element level* errors return a FieldIdx and KeyValue to identify the exact point of failure.

Assume that all parts of an operation before the point of failure completed.

Table 3-14: BlockStatus Codes and Error information

Error Code (*I2O_PARAMS_STATUS*_xxx)	**Error Info content**	**Description**
_SUCCESS	{none}	Normal execution - Operation Results field contains entire results as expected. Size of Error Information field is zero.
_BAD_KEY_ABORT	Operation code, GroupNumber FieldIdx = -1 AdditionalStatus NumberKeys ListOfKeyValues Pad	The KeyValue specified was not recognized by the device. Operation completed up to the first bad KeyValue, in the ListOfKeyValues, in the ErrorInformation structure. More than one bad KeyValue may be supplied, indicating other values that also cause this error.
_BAD_KEY_CONTINUE	Operation code, GroupNumber FieldIdx = -1 AdditionalStatus NumberKeys ListOfKeyValues Pad	The device did not recognize KeyValue specified. Operation completed except for the rows for KeyValues, in the ListOfKeyValues, in the ErrorInformation structure.
_BUFFER_FULL	{none}	The operation result exceeds the available space in the result buffer. Partial completion occurred.

Error Code **(*I2O_PARAMS_STATUS*_xxx)**	**Error Info content**	**Description**
_BUFFER_TOO_SMALL	None	The result buffer was too small for the first operation in a message to provide a ResultBlock. If this occurs, the result list contains a single ResultBlock with this error code. No parameter accesses are performed (i.e., size of Operation Results field is 0).
_FIELD_UNREADABLE	Operation code, GroupNumber FieldIdx AdditionalStatus NumberKeys = 0	A field was unreadable - error in execution. Partial completion up to the first encounter of the identified field occurred.
_FIELD_UNWRITEABLE	Operation code, GroupNumber FieldIdx AdditionalStatus NumberKeys = 0	The field could not be altered. Partial completion up to the first encounter of the identified field occurred.
_INSUFFICIENT_FIELDS	Operation code, GroupNumber	At least one field index must be specified in the operation. No parameter accesses were performed (i.e., size of Operation Results field is 0).
_INVALID_GROUP_ID	Operation code, GroupNumber	The group number supplied did not match any defined group of the device. No parameter accesses were performed (i.e., size of Operation Results field is 0).
_INVALID_OPERATION	Operation code, GroupNumber	An unrecognized operation code was specified in an OperationBlock. No parameter accesses were performed (i.e., size of Operation Results field is 0).
_NO_KEY_FIELD	Operation code, GroupNumber	The key field was not specified as the first field. No parameter accesses were performed (i.e., size of Operation Results field is 0).
_NO_SUCH_FIELD	Operation code, GroupNumber FieldIdx AdditionalStatus NumberKeys = 0	A field index exceeded the range of fields defined for the group. Partial completion up to the first encounter of the identified field occurred.
_NON_DYNAMIC_GROUP	Operation code, GroupNumber	An add/delete/clear operation was attempted on a table group that does not support that operation. No parameter accesses were performed (i.e., size of Operation Results field is 0).
_OPERATION_ERROR	{none}	The OperationBlock could not be parsed. This can result from errors in preceding operations. No parameter accesses were performed (i.e., size of Operation Results field is 0). This error might result from a runt or malformed request. In this case, this is the last reply block and the operation code and group number are not known.

Error Code (***I2O_PARAMS_STATUS***_xxx)	Error Info content	Description
_SCALAR_ERROR	Operation code, GroupNumber FieldIdx AdditionalStatus NumberKeys = 0	An unspecified error occurred on a scalar group. Partial completion up to the first encounter of the identified field occurred.
_TABLE_ERROR	Operation code, GroupNumber FieldIdx AdditionalStatus NumberKeys = 1 KeyValues + Pad	An unspecified error occurred on a table group. Operation completed for all key values up to specified FieldIdx of the specified KeyValue.
_WRONG_GROUP_TYPE	Operation code, GroupNumber	An operation defined only on scalar groups was attempted on a table group, or vice versa. No parameter accesses were performed (i.e., size of Operation Results field is 0).

3.4.7.6 Generic Parameter Group Definitions

The following parameter groups apply to all devices and must be supported by all devices, unless otherwise stated.

Typically, only the primary service user, as established by ***UtilCalm***, modifies a device's operating parameters. However, some parameters are typically set directly by a management agent, rather than through the primary service user. Therefore, management agents do not alter parameters unless expressly allowed. The parameter groups listed in the following tables are exceptions and any parameter identified as read-write may be altered by management.

3.4.7.6.1 The GROUP_DESCRIPTOR Group

To allow discovery at run-time of the parameter groups supported by a device, each device provides the *PARAMS_DESCRIPTOR* group. It describes the device's parameter groups. The rows of this group represent the groups supported by the device. The fields include the GroupNumber, number of fields and rows, and whether or not a group is a table.

Table 3-15: Group F000h - PARAMS DESCRIPTOR Group

GroupNumber	F000h
GroupType	TABLE
Name	***PARAMS_DESCRIPTOR***
Description	This table identifies the parameter groups provided by this device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 Bytes	GroupNumber	The 16-bit identifier of the group
1	r	2 Bytes	FieldCount	The number of fields in the group
2	r	2 Bytes	RowCount	The number of rows currently in the group
3	r	1 Byte	Properties	Bit 0: 0 = group is scalar (i.e. always one row) 1 = group is a table (i.e. zero or more rows) Bit 1: 0 = row addition not supported 1 = row addition supported Bit 2: 0 = row deletion not supported 1 = row deletion supported Bit 3: 0 = clear operation not supported 1 = clear operation supported
4	r	1 Byte	reserved1	

Note that the *PARAMS_DESCRIPTOR* group includes an entry for itself. This provides backward compatibility, allowing additional field definitions at a later date. The management agent may perform *LIST_GET* (GroupNumber=F000h, FieldIdx=-1, KeyValue=F000h) or *SIZE_GET* (GroupNumber=F000h, FieldIdx=-1) to learn which fields this group supports.

Table 3-16: Group F001h - Physical Device Table

GroupNumber	F001h
GroupType	TABLE
Name	***PHYSICAL_DEVICE_TABLE***
Description	A table of all physical devices that correspond to this device. For a DDM, the table lists all adapters assigned to the DDM. For a device registered by the DDM, it is a list of all adapters assigned to the DDM; if the adapter is released, the device will no longer exist.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 Bytes	AdapterID	The Adapter ID of each device as represented in the HRT (See Chapter 4).

Table 3-17: Group F002h - Claimed Table

GroupNumber	F002h
GroupType	TABLE
Name	***CLAIMED_TABLE***
Description	A table of all I_2O devices claimed by this DDM in creating this device. Traversing these tables should yield TIDs for adapters and physical devices that correspond to this device. For a DDM TID, this lists all TIDs assigned to the DDM.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 Bytes	ClaimedTID	The TID for each claimed device (the most significant four bits are zeros)

Table 3-18: Group F003h - User Table

GroupNumber	F003h
GroupType	TABLE
Name	***USER_TABLE***
Description	A table of all users that claim this device. Following this list up to the top identifies all modules affected if the device is reset or removed.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 Bytes	Instance	A unique value assigned by the DDM
1	r	2 Bytes	UserTID	The TID for each user (the most significant four bits are zeros)
2	r	1 Byte	ClaimType	The ClaimType from the Claim message
3	r	1 Byte	reserved1	
4	r	2 Bytes	reserved2	

Table 3-19: Group F005h - Private Message Extensions

GroupNumber	F005h
GroupType	TABLE (optional)
Name	***PRIVATE_MESSAGE_EXTENSIONS_TABLE***
Description	A list of private message extensions that this device supports.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 bytes	ExtInstance	The n^{th} instance of private message extension
1	r	2 bytes	OrganizationID	ID assigned to the organization defining the message extension. See section 3.4.1.6.
2	r	2 bytes	XFunctionCode	Function code extension. See section 3.4.1.6.

Table 3-20: Group F006h - Authorized User Table

GroupNumber F006h

GroupType TABLE

Name ***AUTHORIZED_USER_TABLE***

Description A table of all TIDs authorized to claim this device as an alternate user. Only the primary user may modify this table.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r/w	2 Bytes	AlternateTID	The TID for an authorized alternate user (the most significant four bits are zeros)

Table 3-21: Group F100h - Device Identity

GroupNumber F100h

GroupType SCALAR

Name ***DEVICE_IDENTITY***

Description Information pertaining to the device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 Bytes	ClassID	Multi-field parameter identifies the class of messages that may be sent to this TID.
1	r	2 Bytes	OwnerTID	TID assigned to the module that claimed this resource as the primary user (the most significant four bits are zeros). A value of FFFh indicates the device may be claimed.
2	r	2 Bytes	ParentTID	TID assigned to the DDM or device that created this I_2O device (the most significant four bits are zeros). The parent of a DDM class device is always 000h. The parent of a port is the TID of the DDM. The parent of a peripheral is the TID of the port or, if no port device exists, the TID of the DDM.
3	r	16 Bytes	VendorInfo	ASCII Vendor Information. Returns the vendor information for the device.
4	r	16 Bytes	ProductInfo	ASCII Product Information. Returns the product information for the device.
5	r	16 Bytes	Description	ASCII Product Description. Returns the product description for the device.
6	r	8 Bytes	ProductRevLevel	ASCII revision level of the product. The length of eight characters allows BetaX.XX.
7	r	1 Byte	SNFormat	See section 3.4.4 for definition and values.
8	r	Variable	SerialNumber	See section 3.4.4 for definition and format.

Table 3-22: Group F101h - DDM Identity

GroupNumber	F101h
GroupType	SCALAR
Name	***DDM_IDENTITY***
Description	Information pertaining to the software (controlling DDM)

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 Bytes	DdmTID	TID section assigned to the DDM that registered this device, or registered the port device that registered this device. The most significant four bits are zeros.
1	r	24 bytes	ModuleName	ASCII Module Information. Returns the product information for the device software. (Example: SuperSlow SCSIware2000) All devices created by the same software module report the same information.
2	r	8 bytes	ModuleRevLevel	ASCII revision level of the product. The length of eight characters allows BetaX.XX.
3	r	1 byte	SNFormat	See section 3.4.4 for definition and values. DDM would support the WW ID format.
4	r	12 bytes	SerialNumber	See section 3.4.4 for definition and format. If the software does not contain a serial number, report zeros.

Table 3-23: Group F102 - User Information

GroupNumber	F102h
GroupType	SCALAR
Name	***USER_INFORMATION***
Description	This table contains parameters the user enters to identify devices and their physical location in the system. This information may show up in error reports or configuration screens to help the user relate to the device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r/w	64 Bytes	DeviceName	ASCII user information. (Examples: SCSI Controller, Ethernet Port)
1	r/w	64 Bytes	ServiceName	ASCII user information. (Examples: Right SCSI Enclosure, Test Network)
2	r/w	64 Bytes	PhysicalLocation	ASCII user information. (Examples: System Board, PCI slot 5)
3	r/w	4 Bytes	InstanceNumber	ASCII user information. (Examples: A, 2, or C:)

Table 3-24: Group F103h - SGL Operating Limits

GroupNumber	F103h
GroupType	SCALAR
Name	***SGL_OPERATING_LIMITS***
Description	This group provides attributes that identify SGL operating limits for this device. Exceeding these values may result in rejection of request messages.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 Bytes	SglChainSize	The device performs optimally when the SGL chain list size is less than this number of bytes.
1	r	4 Bytes	SglChainSizeMax	The device cannot handle an SGL chain greater than this size (bytes).
2	r/w	4 Bytes	SglChainSizeTarget	The primary user writes this value to indicate its optimal SGL chain buffer size. If the device can change its current operation to accommodate, it should. Otherwise, if it can change the next time the DDM is loaded, it should remember this value and do so. Reading this value provides the value that the DDM plans to use at next initialization.
3	r	2 Bytes	SglFragCount	The device performs optimally when the number of data fragments is less than this number. Each of the following counts as one fragments: an address in a page list element, each simple SGL element, each immediate data element, and each bit bucket element.
4	r	2 Bytes	SglFragCountMax	The maximum number of buffer fragments that the device can handle. The device cannot handle a message containing more than this number.
5	r/w	2 Bytes	SglFragCountTarget	The primary user writes this value to indicate the optimal number of fragments for a single message. If the DDM can change its current operation to accommodate, it should. Otherwise, if it can change the next time the DDM is loaded, it should remember this value and do so. Reading this value provides the value that the DDM plans to use at next initialization.

3.4.7.6.2 Sensor Parameter Group

System Managers want warning of potential and actual catastrophic system failures. Manufacturers of high-end systems and peripherals design I/O devices with sensors that read and report system faults or environmental issues that arise with their equipment. The Sensor parameter group presents generic attributes that address the data expected from such instrumentation. This option group applies to any class of device incorporating such sensors.

Table 3-25: Group F200h - Sensors

GroupNumber F200h

GroupType TABLE (optional)

Name ***SENSORS***

Description This table contains the parameters necessary to report sensor information and set the appropriate limits for event notification. They are optional and needed only when sensors are implemented and the information is reported across the I_2O shell interface.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 Bytes	SensorInstance	The nth ordering of sensors
1	r	1 Byte	Component	The component where the sensor is located (e.g. Planar logic board, CPU, chassis) 0 Other 1 Planar Logic Board 2 CPU 3 Chassis 4 Power Supply 5 Storage 6 External
2	r	2 Bytes	ComponentInstance	The nth ordering of the component
3	r	1 Byte	SensorClass	Defines the type of pre-processing to be performed on the sensor. Digital sensors have no threshold values or hysteresis. 0 analog 1 digital
4	r	1 Byte	SensorType	Sensor type (and default unit) 0 Other 1 Thermal (°C) 2 DC voltage (DC volts) 3 AC voltage (AC volts) 4 DC current (DC amps) 5 AC current (AC amps) 6 Door open 7 Fan operational
5	r	1 byte (INT8)	ScalingExponent	Decimal exponent of scaling factor E.g.: 0 = unity, 3 = 10^3 (kilo-); -5 = 10^{-5} (10 × micro-). 0 for *digital* sensor class. This attribute should represent the precision with which thresholds and readings should be represented. Note: This really defines where the decimal point is placed in the representation of all the readings and thresholds that follow (i.e. from actual reading through maximum reading). Example: A +5 volt sensor measured in tens of microvolts would exhibit a nominal reading of 500,000 and a scaling exponent of -5.

GroupNumber	F200h
GroupType	TABLE (optional)
Name	***SENSORS***
Description	This table contains the parameters necessary to report sensor information and set the appropriate limits for event notification. They are optional and needed only when sensors are implemented and the information is reported across the I_2O shell interface.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
6	r	4 Bytes (INT32)	ActualReading	The current reading of the sensor, scaled by the *scaling exponent* above. For example, an actual reading of 686 when the scaling exponent is -3 means a DC sensor value of 686×10^{-3} volts, or 686 milli-volts.
7	r	4 Bytes (INT32)	MinimumReading	The minimum reliable reading the sensor can support, scaled by the scaling exponent.
8	r/w	4 Bytes (INT32)	Low2LowCatThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the Low Catastrophic (non-operational) region from the Low (operational) region
9	r/w	4 Bytes (INT32)	LowCat2LowThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the Low (operational) region from the Low Catastrophic (non-operational) region
10	r/w	4 Bytes (INT32)	LowWarn2LowThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the Low (operational) region from the Low Warning (operational) region
11	r/w	4 Bytes (INT32)	Low2LowWarnThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the Low Warning (operational) region from the Low (operational) region
12	r/w	4 Bytes (INT32)	Norm2LowWarnThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the Low Warning (operational) region from the Normal (operational) region
13	r/w	4 Bytes (INT32)	LowWarn2NormThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the Normal (operational) region from the Low Warning (operational) region
14	r	4 Bytes (INT32)	NominalReading	The nominal reading, scaled by the scaling exponent, for the sensor.
15	r/w	4 Bytes (INT32)	HiWarn2NormThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the Normal (operational) region from the High Warning (operational) region
16	r/w	4 Bytes (INT32)	Norm2HiWarnThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the High Warning (operational) region from the Normal (operational) region

GroupNumber	F200h
GroupType	TABLE (optional)
Name	***SENSORS***
Description	This table contains the parameters necessary to report sensor information and set the appropriate limits for event notification. They are optional and needed only when sensors are implemented and the information is reported across the I$_2$O shell interface.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
17	r/w	4 Bytes (INT32)	High2HiWarnThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the High Warning (operational) region from the High (operational) region
18	r/w	4 Bytes (INT32)	HiWarn2HighThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the High Warning (operational) region from the High (operational) region
19	r/w	4 Bytes (INT32)	HiCat2HighThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the High Warning (operational) region from the High Catastrophic (non-operational) region
20	r/w	4 Bytes (INT32)	High2HiCatThreshold	The sensor reading, scaled by the scaling exponent, at which sensed condition enters the High Catastrophic (non-operational) region from the High Warning (operational) region
21	r	4 Bytes (INT32)	MaximumReading	The maximum reliable reading the sensor can support, scaled by the scaling exponent
22	r	1 Byte	SensorState	0 Normal 1 Abnormal 2 Unknown 3 Low Catastrophic (LoCat) 4 Low (Low) 5 Low warning (LoWarn) 6 High Warning (HiWarn) 7 High (High) 8 High Catastrophic (HiCat)
23	r/w	2 Bytes	EventEnable	Generate an event when a change in state occurs for any of the indicated bit masks: Bit 0: operational state change (digital sensors only) Bit 1: low catastrophic Bit 2: low reading Bit 3: low warning Bit 4: change back to normal from out of range state Bit 5: high warning Bit 6: high reading Bit 7: high catastrophic

Location

Location of sensors is comprehended in attributes 0-3. Attribute 0, *sensor instance*, indexes the group table for devices that have multiple sensors. Attribute 1, *component instance*, describes the component where a sensor resides. Attribute 2, *component instance*, provides differentiation for sensors on devices that may have multiple instances of a particular component type. Attribute 3, *sensor class*, describes whether the sensor provides a quantifiable value (0=analog) or a simple binary status (1=digital).

Nominal Reading

The *nominal reading* of a sensor is comprehended by attribute 15. Attribute 15, nominal reading, is the mantissa of the nominal value that the sensor might expect. This attribute reports a INT32 value that must be multiplied by the precision reported in attribute 6.

Scaling and Precision

Scaling and precision of sensors are comprehended together by Attribute 5, *scaling exponent*. The scaling exponent reports the precision to which the sensor can report meaningful data. The value is the decimal exponent that represents the least-significant decimal digit of the reportable value. For example, a value of -3 means that scaled attributes' (Attributes 7-22) values must be multiplied by 10^{-3} to obtain the actual sensor value.

Sensor Range

The operational range of the sensor is reported by attributes 8 and 22. Attribute 8, *minimum reading*, reports the minimum reliable reading that the sensor supports. Attribute 22, *maximum reading*, reports the maximum reliable reading that the sensor supports. These attributes report an INT32 value that must be multiplied by the precision reported in attribute 5.

Actual Reading

The actual reading of the sensor is reported in attributes 4, 5, and 7. Attribute 4, *sensor type*, reports the type of information the sensor reports. Units of measure are implied by the type of sensor reported. (See group definition table.) Attribute 7, *actual reading*, is an INT32 value that represents the mantissa of the value reported by the sensor. Attribute 5, *scaling exponent*, reports the decimal exponent of the *actual reading* attribute. For example, a scaling exponent value of -3 means that the actual reading attribute value must be multiplied by 10^{-3} to obtain the actual attribute value.

Thresholds

To avoid the need to regularly poll sensors to determine system health, some mechanism must produce meaningful exceptions when sensor readings change significantly. Attributes 9-14 and 16-21 report thresholds that define up to seven sensor states.

The boundary of each state is defined by two thresholds, one for each direction of state transition. Figure 3-63 illustrates each state with a region of hysteresis (determined by rising and falling threshold values) between each. This provides a mechanism for filtering low-level, non-monotonic sensor behavior near the threshold region. It also provides a way to comprehend the error induced by sensor tolerances.

Status

The status of the sensed parameter's value, regarding the thresholds described in Attributes 9-14 and 16-21, can be determined by the value of Attribute 23, *state*. This provides a simple method for an exception handler on the host to determine why an exception was generated.

Exception Generation

State-change exceptions are based on a mask represented in Attribute 24. The mask value is determined by setting to 1 each bit corresponding to a state. An exception is generated upon entry into the state corresponding to each bit set in the mask.

3.4.8 User Interactive Configuration Dialogue

The basic configuration parameter group access facilities of the previous section provide a mechanism for clients to query and modify device parameters. This section describes a facility that allows the DDM to create interactive dialogues with a human operator to display and modify parameters in those groups.

The purpose of the dialogue facility is to have a DDM-defined and controlled communication with a human operator. The facility is self-contained in a downloaded DDM and is available in any I_2O-enabled system. The facility does not depend on any DDM-specific program or files not contained in the downloaded DDM, which might get separated from the DDM or have version mismatches. The facility allows DDM-defined presentation of data and command interaction, as opposed to a generic front end such as a DMI management application. The facility is NOS independent and does not rely on DDM-specific front-end applications that must be ported to each NOS. Even though the vendor may provide alternatives, the dialogue facility provides a fundamental configuration and monitoring facility.

To meet these goals, the dialogue facility consists of the following:

- a host-to-IOP dialogue protocol, based on HTML
- a TCL-subset interpreter, to facilitate generating the HTML pages and parsing input
- commands to access device parameter groups from the TCL scripts.

3.4.8.1 Configuration Dialogue Messages

As a basis for the I_2O dialogue facility, HTML appropriately combines simplicity and expressiveness. It is also becoming ubiquitous as a document format. The ***UtilConfigDialog*** message (defined in Chapter 6) provides a mechanism for a DDM-directed dialogue with a human operator, based on the HTML protocol.

The request message contains the number of a dialogue page, any form data being returned, and a buffer where the device places the reply. The reply contains HTML text that the host presents to the human operator via an HTML viewer, such as a Web browser. The form data is text typically generated from an HTML form submitted with an HTTP POST. The text is in the form field1=value1&field2=value2, and usually represents new values for selected fields in selected groups.

Every device must supply a page number 0, the device’s home page. The host requests this page when a device sets the dialogue request indication in the LCT. Other pages are typically accessed by HTML links.

3.4.8.2 Host-side Dialogue Components

A Web browser is the obvious viewer for the HTML dialogue pages. Browsers are ubiquitous and can be operated remotely from the information provider. However, Web browsers use the HTTP protocol for accessing pages. Thus, to use unmodified browsers for interacting with I_2O dialogue pages, a *transducer* must convert HTTP messages to I_2O messages. This transducer may be implemented in a number of ways; one usually provides an I_2O server application using the CGI protocol to an HTTP server. This results in the following configuration:

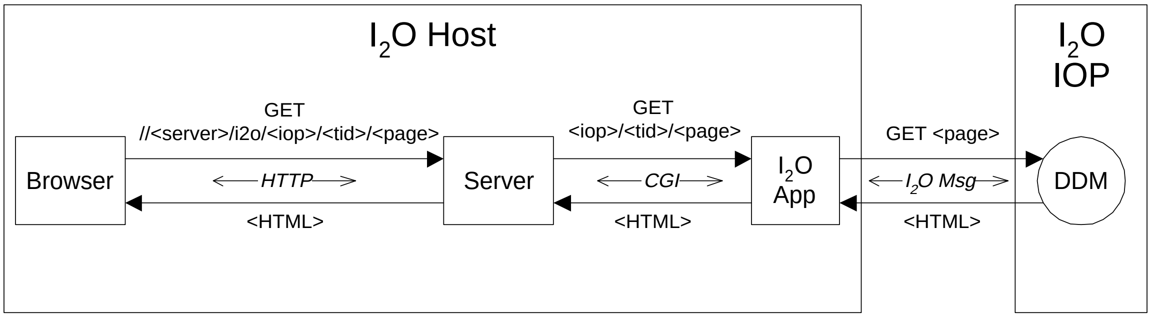


Figure 3-64. Typical Host Configuration Dialogue Mechanism

In this configuration, a standard browser sends HTTP GET messages to a standard server with a URL that identifies the following:

- server (e.g. localhost or a TCP/IP address)
- I_2O application to be invoked by the server via CGI (e.g. i_2o)
- specific IOP to address (e.g. an IOP number on that host)
- specific TID to be addressed within that IOP
- the page number requested.

Based on this URL, the server invokes the I_2O application, passing the rest of the URL containing the IOP number, TID, and page number. The I_2O application translates this information into an ***UtilConfigDialog*** message, sending it to the specified TID on the specified IOP. The HTML text is extracted from the reply and returns to the server, which returns it to the browser for display.

In an HTML form incorporating menus, check boxes, and radio buttons, the user can click a *submit* button. The browser then sends the server an HTTP POST message that contains a URL, followed by the data from the form in text as:

field1=value1&field2=value2…

In this case, the server forwards the remainder of the URL and form data to the I_2O application, which translates this into an ***UtilConfigDialog*** message containing the form data. Thus, both HTTP GET and POST messages get translated into an ***UtilConfigDialog*** message.

Every device must supply a home page (page 0), so, using an IOP number and a TID, the URL for the home page of a particular device is easily constructed. The IOP executive (TID 000h) also supplies a home page, so given an IOP number, the URL for the home page of that IOP is easily constructed. The IOP dialogue pages link to the home pages of all registered devices. Also, each device home page should provide the reverse link to the IOP home page.

These links, and those within a device’s pages, are considered *relative* URLs, or just the page number, so devices do not need to construct full URLs with IOP number, TID, and so forth.

Any devices that currently request a configuration dialogue must be highlighted on the IOP’s home page, and their links displayed in the same order as their entries in the LCT.

Finally, the I_2O CGI application can also construct a *master* I_2O page that links to the IOP home pages for each IOP on a given server. It queries the NOS for the IOP numbers of active IOPs.

Note that the browser may be on a machine remote from the server.

3.4.8.3 TCL Scripts

To interact with a human user using the HTML-based dialogue facility, a device must format HTML text and parse input *form* data. While this can be done in the C language, it is far easier to use text macro and scripting facilities. The IRTOS provides a scripting facility described in Chapter 5.

4
I_2O Shell Interface Specification

The two perspectives of the IOP are the system view, or shell, and the internal view, or core. This chapter discusses the system view, which is how the host and other IOPs see an IOP. In this context, the IOP is a service provider, and it abstracts the service provided by its I/O devices. Each IOP presents a common programming interface to the host and the other IOPs. This system programming interface has three components:

1. a register-level interface
2. the protocol for exchanging messages
3. a set of executive-class messages

4.1 Overview of the I_2O Shell Operation

The primary function of the IOP shell is providing the abstract interface for its I/O devices and thus establishing a communication channel between its DDMs and drivers on the host and other I/O platforms. An IOP abstracts its own embedded devices and can also be the host for drivers and adapters provided by another vendor. The capability to host such loadable modules is optional and is described in detail in the core specification (see Chapter 5).

This section specifies the mechanism for loading or installing a driver on an IOP. Some applications for an IOP might not provide any physical expansion capability to support additional adapters, and thus these IOPs will not support loadable HDMs. Even in this case it is still beneficial to support loading third-party ISMs. This document makes no assumptions about the resources necessary to store and execute third-party drivers. Due to resource limitations, each IOP limits the number and size of loadable modules it can support. An IOP that cannot install or load an additional module simply rejects the request.

The I_2O shell specification supports the following:

- **IOP initialization** initializes the inbound message queues, resets the outbound message queues, and builds a logical configuration table describing current I/O devices behind the IOP.
- **IOP configuration** establishes a system configuration table describing all IOPs in the system, installs and loads DDMs, assigns adapters to HDMs, assigns devices to ISMs, and builds an external connection table as connections are established with other IOPs.
- **Initiation of DDM installation** directs the core to install the DDM. DDM installation is described in the core specification.
- **DDM load** loads the DDM code, invokes its initialization process, attaches adapters and devices that the module will control, and enables the DDM configuration process.
- **Setup for DDM configuration** assigns adapters and devices to the DDM and updates the IOP's physical configuration table. The DDM configuration is described in the core specification.
- **Message service** routes messages and delivers them to the appropriate driver, creates connections, and delivers messages on those connections
- **Transport service** moves blocks of raw data between system memory and local memory.

4.2 IOP System Interface

An IOP has a register-level system interface that implements queues for sending and receiving messages.

4.2.1 Register-level Interface

Messages reside in a shared memory structure, the message frame. A message frame is indicated by its *message frame address* (MFA). The MFA specifies the first byte of the message frame header. This MFA is actually the offset between the start of the target messenger's shared memory and the start of the message. Messages are passed by indicating the frame's MFA to the target messenger instance (i.e., an IOP or the host). Each IOP contains hardware to efficiently pass MFAs between messenger instances.

4.2.1.1 IOP Message Unit

The IOP supplies two paths for messages, an inbound queue to receive messages from the host and all other IOPs, and an outbound queue to pass messages to the host. Each queue is implemented as a pair of FIFOs, as shown in Figure 4-1. The inbound message queue contains a pair of hardware FIFOs, Free_List and Post_List, for allocating and posting messages to an IOP. Another pair of FIFOs comprises the outbound message queue for messages to the host.

A FIFO is a buffer with two interfaces. One interface fills the buffer and the other drains it. Message frames may be placed in a FIFO in any order. The drain removes the oldest entry first. The Free_List FIFO holds the empty message frames and the Post_List FIFO holds the active message frames. A sender (message producer) draws from the head of the Free_List and posts to the tail of the Post_List. A recipient (message consumer) draws from the head of the Post_List and releases by posting to the tail of the Free_List.

For the inbound queue, the host and other IOPs are the message producers and the IOP containing the FIFO is the message consumer. The host or other IOP is not required to post message frames in the same order that the frames were allocated. For the outbound queue, the local IOP (i.e., the IOP providing the outbound FIFO) is the message producer and the host is the message consumer. The host is not required to release message frames in the same order that they were retrieved.

4.2.1.2 Message Queuing

For the inbound queue the IOP provides, the sender can read a particular system memory location to retrieve the MFA of the next empty message frame from the Free_List. When the sender deposits its message in the message frame, it writes the frame's MFA to the same system memory location. This causes the MFA to be placed on the end of the Post_List. The IOP removes an MFA from the Post_List, processes the message, and then places the MFA (now an empty message frame) on the end of the Free_List. If the Free_List is empty when the sender reads the Free_List FIFO, the IOP must supply the value FFFF-FFFFh. This mechanism is used by all messengers (host and other IOPs) to send messages to the local IOP.

When sending messages to another IOP, the local IOP uses the inbound queue of the remote IOP.

The IOP provides an outbound queue for sending messages to the host. The host OS reads a particular system memory location to retrieve the MFA of the next message from the Post_List.

When the host has consumed the message, the host writes the MFA to the same system memory location, which causes the MFA to be placed on the end of the Free_List. The IOP removes an MFA from the Free List, fills the frame with a message, and then places the MFA on the end of the Post_List. If the Post_List is empty when the host reads the Post_List FIFO, the host receives the value FFFF-FFFFh.

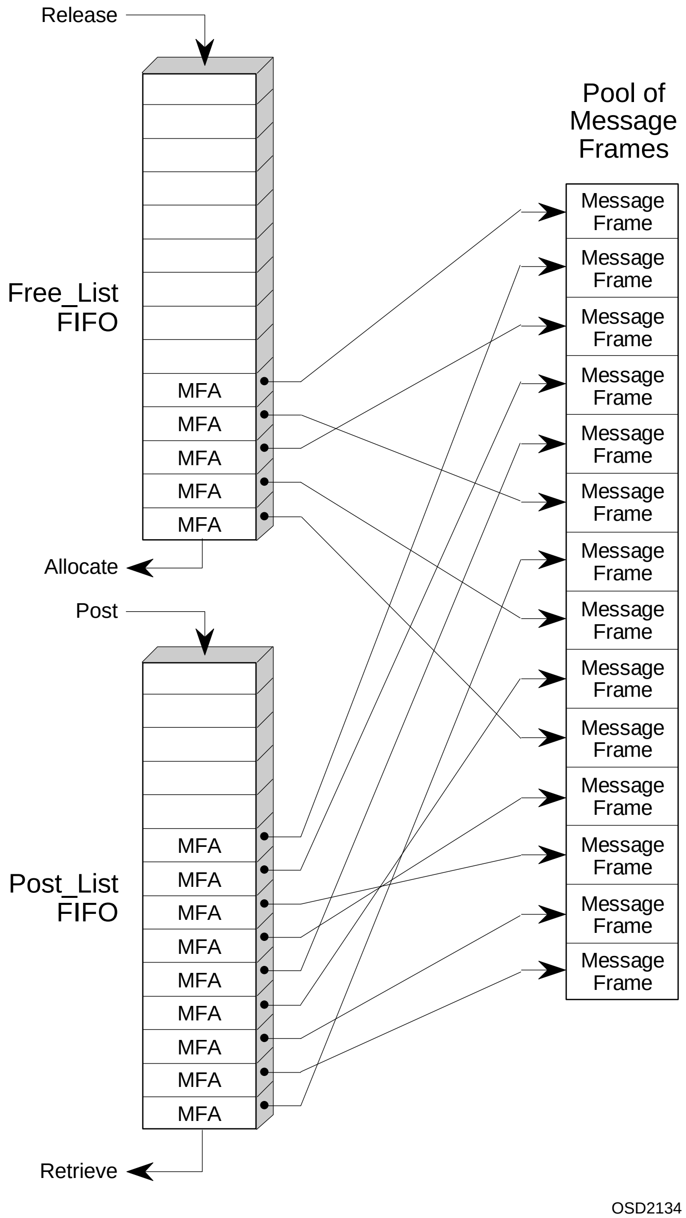


Figure 4-1. Message Queue Example

4.2.1.3 Initializing the Queues

Until the IOP initializes its inbound queue, reading the inbound FIFO must not return any value other than FFFF-FFFFh. Any time a value other than FFFF-FFFFh is read, the inbound FIFO must be ready to receive the posting of that MFA.

The IOP initializes its inbound queue by creating a number of message frames in its shared memory and placing the MFA of each in the Inbound Free_List FIFO. Message frames must start on a 32-bit boundary.

All of the MFAs can be in either the Free_List FIFO or the Post_List FIFO. To prevent overflow, the total number of message frames created for a queue must be less than the size of either FIFO.

The system memory location for the inbound and outbound FIFOs can be discovered as discussed in section 4.2.1.5 *System Bus Extensions* below. The host uses this interface to initialize the IOP's outbound queue. The host builds a number of message frames and then posts them to the Free_List by writing their MFAs to the outbound FIFO. These message frames are also required to start on a 32-bit boundary. Because the host claims all of system memory as its domain, the outbound MFA is the offset of the message from 0, and thus the physical address of an outbound message is its MFA. This is not the case for inbound message frames.

Each FIFO must support a minimum of 16 message frames. The memory of each message frame must be contiguous, but all the message frames need not be in one contiguous block of memory. All message frames for a queue are the same size. The minimum size of a message frame is 64 bytes, and it must start on a 32-bit boundary.

4.2.1.4 System Interrupt Generation

IOP facilities generate and mask system interrupts that indicate events that require immediate attention, such as posting messages to the outbound queue. The facilities for managing the interrupt from an IOP allow the host to completely disable the interrupt and resort to a polled mode of operation. At times, it is desirable to temporarily disable the interrupt and schedule its processing at an appropriate time, after which the interrupt is again enabled. The following interrupt facilities serve these purposes:

- In the *Interrupt Status Register*, each bit represents the source of an interrupt. That bit is set when the source requests service. One of the bits represents the outbound post list. That bit is set any time one or more MFAs can be retrieved by the host. Writing to this register has no effect on the value of this bit.
- The *Interrupt Mask Register* corresponds to the interrupt status register and masks the interrupt source from generating a system interrupt. Each bit in this register corresponds with a bit in the Interrupt Status Register; setting the mask register bit disables that source from causing a system interrupt. The value of the Interrupt Mask Register does not affect the value in the Interrupt Status Register. Therefore, the IOP generates a system interrupt only if the value of the Interrupt Status Register, logically ANDed with the inverse of the Interrupt Mask Register, produces a non-zero result. The default mask register is set to all ones, or no interrupts enabled.

4.2.1.4.1 Operation

For polled operation, the host leaves the mask register set to all ones. The host determines if messages are waiting by reading either the Outbound Post_List FIFO or the Interrupt Status Register. For interrupt-driven operation, the host enables interrupts by clearing the Outbound Post List bit in the Interrupt Mask Register. The host ISR disables interrupts at the source by setting that bit. Again, the host can either read the Outbound Post_List FIFO or the Interrupt Status Register to determine if messages are present. Typical ISR operation is that the host:

(1) reads the Interrupt Status Register to verify the interrupt source

(2) disables the interrupt at the source, if deferred processing is necessary

(3) processes the FIFO until all messages are retrieved

(4) enables the interrupt source.

4.2.1.5 System Bus Extensions

This section focuses on the operating characteristics for different system buses. The primary concern is the capability of the host OS to locate an IOP and thus determine its FIFO addresses. That allows it to send and receive messages. For the host to discover and form a connection with an IOP, these attributes are necessary:

- a method to uniquely identify the presence of each IOP
- the location of its inbound Free_List FIFO
- the location of its inbound Post_List FIFO
- the location of its outbound Free_List FIFO
- the location of its outbound Post_List FIFO
- the location of its shared memory, so the host can interpret MFAs properly.

4.2.1.5.1 Extensions for PCI

The host identifies an IOP by its PCI class code. Refer to [PCI] for further detail. The PCI class code has three fields:

1. Base Class

2. Subclass

3. Programming Interface

The Base Class value assigned for intelligent I/O controllers is 0Eh. The Subclass code assigned for PCI devices conforming to the I_2O specification is 00h. Two programming interfaces are specified: Both employ 32-bit data width, 32-bit addressing, and little endian byte order, with the following characteristics:

4.2.1.5.1.1 Programming interface code value = 00h

- Name = hardware FIFO at offset 40h
- Location of the inbound Free_List FIFO at memory offset 40h in the memory region specified by the first base address configuration register indicating *memory space* (offset 10h, 14h, and so forth). The inbound Free_List FIFO is a read-only register.
- Location of the inbound Post_List FIFO at memory offset 40h in the memory region specified by the first base address configuration register indicating *memory space* (offset 10h, 14h, and so forth). The inbound Post_List FIFO is a write-only register. This write-only location overlaps with the read-only location of the inbound Free_List FIFO.

- Location of the outbound Free_List FIFO is at memory offset 44h in the memory region specified by the first base address configuration register indicating memory space (offset 10h, 14h, and so forth). The outbound Free_List FIFO is a write-only register.
- Location of the outbound Post_List FIFO at memory offset 44h in the memory region specified by the first base address configuration register indicating *memory space* (offset 10h, 14h, and so forth). The outbound Post_List FIFO is a read-only register. This read-only location overlaps with the write-only location of the outbound Free_List FIFO.
- Each inbound MFA is the offset between the start of the memory region specified by the first base address configuration register and the start of the message.
- Does not support standard interrupt capability.

4.2.1.5.1.2 Programming interface code value = 01h

Same as above, plus system interrupt capability as follows:

- 32-bit memory mapped Interrupt Status Register, at offset 30h in the memory region specified by the first base address configuration register indicating memory space (offset 10h, 14h, and so forth). Bits 0-2 and 4-31 are reserved, and bit 3 is the outbound post list service request.

- 32-bit memory mapped Interrupt Mask Register at offset 34h in the memory region specified by the first base address configuration register indicating memory space (offset 10h, 14h, and so forth), with bit 3 masking the outbound post list service request.

4.2.1.5.2 Extensions for Other Bus Types

Because PCI is the predominant bus in new server designs, this initial version of the specification focuses on PCI. This does not preclude other bus types, but defining extensions for other bus types is left until support becomes necessary.

Note:

These bus extensions apply to the bus where the IOP resides, not on the buses behind an IOP. An IOP may incorporate a number of different expansion buses, as specified in Chapter 5.

4.2.2 Queuing Model

- **Inbound queue.** A message frame is allocated to a message producer when the producer reads the IOP's inbound port. The value it reads is the MFA. The value FFFF-FFFFh signifies an empty list; otherwise, the producer builds or copies its message into the specified message frame. The message is then posted to the IOP by writing the MFA to the IOP's inbound port. The IOP processes the message, and when it finishes with the message frame, places the MFA back on the free list for another messenger to use.

  The message producer is not required to post MFAs to the inbound port in the same order that they were allocated, nor is the IOP required to return MFAs to the free list in the same order they were posted to the post list. In fact, the IOP need not return the message frame to the free list, as long as it can maintain a constant supply of message frames.

  The message producer cannot directly return an MFA to the free list. To return an unused message frame, the message producer uses the ***UtilNOP*** function and posts the MFA to the target's inbound port (see *Utility Messages* in Chapter 6).

- **Outbound queue.** Message frames for sending messages to the host are allocated to an IOP by the host when it writes the frame's MFA to the IOP's outbound port. When the IOP produces a message for the host, the IOP:

  1. pulls an MFA from the outbound free list
  2. places its message in the specified message frame
  3. posts the MFA to the outbound post list FIFO.
  4. The host retrieves a message by reading its MFA from the IOP's outbound port.

     The value of FFFF-FFFFh signifies that the list is empty. Once the host consumes the message, it returns the message frame by writing its MFA to the IOP's outbound port.

     The IOP is not required to post messages to the host in the same order they were allocated, nor is the host required to return MFAs to the free list in the same order they were posted to the post list. In fact, the host need not return that same message frame to the free list, as long as it can maintain a constant supply of message frames.

4.2.3 IOP State

The state of an IOP is characterized by its ability to send and receive messages. The states are defined in Table 4-1, followed by a detailed description of each state. A state table is provided in Table 4-2.

Table 4-1. IOP State Definitions

State Name	State Definition
FAIL	Failed: Non-recoverable operational (software) error occurred. Reset or restart needed.
FAULT	Faulted: Hardware failure detected that prevents operation
HOLD	Quiesced: Accepting only system requests. Outbound queue in operation. System table not valid, external connection table cleared.
INIT	Initializing: Inbound message queue not available
OP	Operational: Accepting requests from system and peers.
READY	Quiesced: Accepting only system requests. Outbound queue in operation. System table valid.
RESET	Initialized: Inbound message queue available for system requests, outbound message queue empty. System table not valid; external connection table cleared.

Detailed description of the IOP State in natural sequence:

INIT state

When an IOP is powered on or reset, it may temporarily enter the *INIT* state while it prepares its inbound message queue (clears the post FIFO and primes the free list) and clears its outbound message queue (clears both the Free_List and the Post_List FIFOs). During the *INIT* state, the IOP cannot receive messages; any attempt to access the free list or post a message is indeterminate. The IOP automatically transitions to the *RESET* state when it finishes its initialization.

RESET state

As soon as the IOP has initialized and its inbound message queue is operational, it enters the *RESET* state. In the *RESET* state, the IOP configures its drivers and builds its logical configuration table. Then it discards any messages except an ***ExecStatusGet*** or an ***ExecOutboundInit***. The ***ExecStatusGet*** does not cause the IOP to change state. The ***ExecOutboundInit*** moves the IOP to the *HOLD* state. See section 4.4 for executive class messages.

HOLD state

In the *HOLD* state, the IOP responds only to executive class messages from the host and discards all others. The ***ExecSysTabSet*** message causes the IOP to move to the *READY* state. An ***ExecIopReset*** message causes a transition back to the *INIT* state.

READY state

In the *READY* state, the IOP responds only to executive class messages from the host, defers processing executive class messages from other IOPs, and discards all other messages. The ***ExecSysEnable*** message causes the IOP to transition to the operational (*OP*) state, at which time the IOP processes deferred messages. An ***ExecIopReset*** message causes a transition to the *INIT* state, in which case deferred messages are discarded. An ***ExecIopClear*** or ***ExecSysModify*** message causes a transition to the *HOLD* state and also causes deferred messages to be discarded.

OP state

In the operational state, the IOP responds to all messages. An ***ExecSysModify*** or ***ExecIopClear*** message causes a transition to the *HOLD* state. An ***ExecIopReset*** message causes a transition to the *INIT* state. A ***ExecSysQuiesce*** message moves the IOP to the *READY* state.

FAIL state

The IOP enters the *FAIL* state any time it detects an inability to operate reliably (e.g., parity error) and needs to restart. If fault notification is enabled and the IOP can, it replies to outstanding ***UtilEventRegister*** messages (see *UtilEventRegister* in Chapter 6). The IOP remains in the *FAIL* state until it receives an ***ExecIopReset*** or ***ExecIopClear*** message from the host. All other messages are discarded. An ***ExecIopReset*** message causes a transition to the *INIT* state. An ***ExecIopClear*** message causes a transition to the *HOLD* state.

FAULT state

The IOP enters the *FAULT* state when it detects a hard failure that prevents it from functioning. If fault notification is enabled and the IOP can, it replies to outstanding ***UtilEventRegister*** messages. While in the *FAULT* state, the IOP discards all messages and curtails all system activity. If the IOP detects that the fault condition has been removed, it transitions to the *FAIL* state.

The state table for the IOP is illustrated in Table 4-2. The relative events are listed in the left column. The intersection of the event row and the current state column produces the action expected of the IOP when the event occurs while the IOP is in that state. A shaded area with the new state's name in **BOLD UPPERCASE** letters indicates state changes. The notes below the table specify other actions.

Table 4-2. IOP State Table

	Current State						
Event	**INIT**	**RESET**	**HOLD**	**READY**	**OP**	**FAIL**	**FAULT**
Hard Reset	-	**INIT**	**INIT**	**INIT**[1,2]	**INIT**[1,2]	**INIT**[1,2]	**INIT**[1,2]
IOP Initialized	**RESET**	n/a	n/a	n/a	n/a	n/a	n/a
ExecStatusGet message	?x?	resp	resp	resp	resp	resp?	resp?
ExecOutboundInit message	?x?	**HOLD**	resp	resp	resp	resp?	discard
ExecSysTabSet message	?x?	discard	**READY**	resp	resp	discard	discard
ExecSysEnable message	?x?	discard	rej	**OP**	rej	discard	discard
ExecIopReset message	?x?	**INIT**[1,2]	**INIT**[1,2]	**INIT**[1,2]	**INIT**[1,2]	**INIT**[1,2]	discard
ExecIopClear message	?x?	discard	resp[2]	**HOLD**[2]	**HOLD**[2]	**HOLD**[2]	discard
ExecSysModify message	?x?	discard	resp[2]	**HOLD**[2]	**HOLD**[2]	**HOLD**[2]	discard
ExecSysQuiesce message	?x?	discard	rej	resp	**READY**	discard	discard
Other Host Exec Class message	?x?	discard	resp	resp	resp	discard	discard
Other IOP Exec Class message	?x?	discard	discard	defer	resp	discard	discard
Non Exec Class messages	?x?	discard	discard	discard	resp	discard	discard
Soft Failure	retry	**FAIL**	**FAIL**	**FAIL**	**FAIL**	-	-
Hard Failure	**FAULT**	**FAULT**	**FAULT**	**FAULT**	**FAULT**	**FAULT**	-
Hard Failure Repaired	n/a	n/a	n/a	n/a	n/a	n/a	**FAIL**

{**BOLD CAPS**} New state

[1] Clear and rebuild logical configuration table

[2] Clear external connection table

?x? Indeterminate - message can be discarded or take action specified in RESET state. Message frame might be lost since it cannot be placed on free list without jeopardy.

defer Do not process message until after ***ExecSysEnable*** message changes state to OP, if any other state change, then discard.

discard Ignore message, return message frame to free list.

n/a Not applicable.

resp Respond, no state change.

resp? Respond if able, no state change.

rej Reject message for cause.

4.3 Programming Model

The programming interface is a message-passing interface. Messages are either requests or replies. The host generates requests, and the IOP processes them and generates replies. A request is never sent to the host. A reply is sent only in response to a request, but there is not necessarily a one-to-one correspondence between requests and replies. Each request contains an Initiator Context field, which is returned unchanged in each reply to that request. The initiator context allows the host to route the reply and associate it with a request.

Messages fall in one of two categories: *executive* or *I/O transaction* messages. Executive messages are between messenger instances, and manage the I$_2$O system. I/O transaction messages target DDMs for managing an I/O resource. Both executive and I/O transaction messages use the message frame structure and format described in Chapter 3. Even though the same utility messages are defined for every class, many simply do not apply to the executive class (e.g., ***UtilClaim***). The IOP rejects inappropriate utility messages as “function not supported”.

Each I/O driver module registers a number of I_2O devices. Each device has an associated class and a TID. This TID is used for routing messages between modules.

Each IOP assigns a unique TID to devices within its domain and provides that information to other messengers, in a logical configuration table. TIDs route messages to the proper device (i.e., instance of a driver).

Each TID is registered as a particular class and a request to that TID must conform to the rules for its class. The protocol for exchanging messages is discussed below. The protocol for a request is described under the class specification for the device. The protocol for processing executive class messages is specified in Section 4.4.

TID values 000h through 007h are reserved for special use and must not be assigned to devices by the IOP. TID values must be unique within the IOP. Since all IOPs independently assign their own TIDs, TIDs are not unique across the system. Thus, peer communication between messengers requires an alias. An alias is the TID value that an IOP assigns to uniquely identify a TID of another IOP. These alias TIDs do not show up in the logical configuration table, but in the IOP’s external connection table.

4.3.1 Special TID Values

Two TID values serve special purposes:

1. **TID=000h** is assigned to the IOP’s executive function. Use of this value in the InitiatorAddress field of a request is limited. This value must appear only in certain executive messages.
2. **TID=001h** is the alias for all OSMs. This value must not appear in the TargetAddress field of a message posted to an IOP, or in the InitiatorAddress field of a request from an IOP. A reply to InitiatorAddress = 001h is always placed in the IOP’s outbound queue.

4.3.2 Host/IOP Communication

The host has a separate and distinct queue for messages from each IOP (i.e., the IOP’s outbound queue), so assigning an alias to uniquely identify the source module is not necessary. Therefore, the host uses the TIDs that the IOP assigns.

All host requests use TID=001h in the InitiatorAddress field and the TID assigned by the target IOP in the TargetAddress field. The reply is placed in the IOP’s outbound queue, with TID=001h in the InitiatorAddress field, and the value from the request’s TargetAddress field in the reply’s TargetAddress field.

Executive class requests from the host always specify TID=000h in the TargetAddress field; executive class replies to the host always specify TID=000h in the TargetAddress field.

4.3.3 Peer Communication

The TargetAddress in a posted request message identifies its destination within the target IOP. When a request enters the target IOP’s inbound queue, the TargetAddress field contains the TID of the target device assigned by the target IOP, and the InitiatorAddress field contains the alias TID that the target IOP assigned to identify the module sourcing the message.

When the target IOP replies to the originator, it must use the TIDs assigned by the originator in both the TargetAddress and InitiatorAddress fields. That is, the reply's InitiatorAddress field contains the TID assigned to the original device by the original IOP; the TargetAddress field contains the alias TID assigned by the original IOP identifying the module sourcing the reply. Therefore, a pair of aliases must exist before messages can pass between peer modules. These aliases are conveyed by the ***ExecConnSetup*** request and reply messages.

4.4 Executive Messages and Structures

Executive class messages communicate between Messenger Instances. Table 4-3 provides a brief description of the base set of executive messages. Function codes 00h through 1Fh are utility messages, generic to all classes. Utility messages are specified in Chapter 6. Not all utility class messages apply to the Executive class.

Table 4-3. Executive Class Messages

Mnemonic	Description
ExecAdapterAssign	Assign an adapter to the specified HDM.
ExecAdapterRead	Request that the IOP read the registers of a hidden adapter.
ExecAdapterRelease	Revoke the adapter assignment.
ExecBiosInfoSet	Indicate a device accessible via BIOS function call – sets field in logical configuration table.
ExecBootDeviceSet	Indicate device used to boot the OS – set field in logical configuration table.
ExecConfigValidate	Notify the IOP that suspect drivers are acceptable.
ExecConnSetup	Establish aliases for sending messages between I_2O devices on different IOPs.
ExecDdmDestroy	Terminate local DDM operation - release all assigned adapters and I_2O devices; destroy all devices created (registered) by the specified module.
ExecDdmEnable	Release ExecDdmQuiesce state and resume normal operation with specified DDM.
ExecDdmQuiesce	Stop sending messages to specified remote DDM (on another IOP) and ignore messages from that DDM. Used when shutting down the other DDM.
ExecDdmReset	Clear all connections with specified DDM. Sent when reloading the DDM.
ExecDdmSuspend	Suspend local DDM operation - quiesce all devices created (i.e., registered) by the specified module.
ExecDeviceAssign	Assign a device to the specified ISM (i.e., invite a connection between the ISM and the device).
ExecDeviceRelease	Release device - break connection.
ExecHrtGet	Request the IOP's hardware resource table.
ExecIopClear	Abort all pending requests without replying. Rebuild inbound message queues and delete all entries in external connection table.
ExecIopConnect	Establish aliases for sending messages between IOP executives.
ExecIopReset	Abort all pending requests without reply. Rebuild IOP environment - reload IRTOS and resident DDMs.
ExecLctNotify	Request the IOP's logical configuration table after next configuration change. When the target IOP modifies its logical configuration table, it replies to this message, sending its logical configuration table (i.e., broadcasting) to everyone who made this request.

Mnemonic	Description
ExecOutboundInit	Clear outbound message queues to their initial (empty) state.
ExecPathEnable	Release PathQuiesce state and resume normal operation with specified IOP.
ExecPathQuiesce	Stop sending messages to specified IOP and ignore messages from that IOP. Used when shutting down the other IOP. Sent before resetting the other IOP.
ExecPathReset	Clear all connections with specified IOP. Sent when resetting the other IOP.
ExecStaticMfCreate	Create and stuff a static message frame.
ExecStaticMfRelease	Release a static message frame.
ExecStatusGet	Return IOP status: state, size of message frames, and size of inbound and outbound message queues.
ExecSwDownload	Download a software module to the IOP.
ExecSwRemove	Delete a software module from IOP's local store.
ExecSwUpload	Upload a software module from the IOP.
ExecSysEnable	Release ExecSysQuiesce state and resume normal operation.
ExecSysModify	Stop sending messages and ignore all but system messages. Also, suspend all activity to adapters on the system bus, in preparation for a physical system configuration change. Especially useful when the host is about to change PCI configuration (e.g., physical address of this IOP or adapters it might control).
ExecSysQuiesce	Stop sending messages and ignore all except system messages. Used to shut down the receiving IOP. Especially useful when the host is about to change PCI configuration (e.g., change physical address of one or more IOPs).
ExecSysTabSet	Provide system configuration table and enable peer operation.

All Executive class messages are single transaction messages. Typically the sender sets the MessageFlags field for requests to 00h (for 32-bit context size) or 02h (for 64-bit context size). For a normal reply, the MessageFlags field should contain C0h (for 32-bit context size) or C2h (for 64-bit context size). Since some requests provide an SGL, the VersionOffset field depends on the location of the SGL. Since all replies are single transaction, the VersionOffset field should be 01h for all replies.

4.4.1 Replies to Executive Class Messages

The reply to each request message can contain a unique payload and depends on the function. In this case, the reply structure is specified immediately following the definition of the request structure. The default reply structure is used when a specific payload is not required. In addition to the normal reply to a request, the messenger may respond to a request when it encounters certain transport failures or cannot process the message.

4.4.1.1 Default Reply

The default reply structure is the single transaction reply template shown in Figure 4-2. It applies to all message functions, unless otherwise indicated.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
Function	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
ReqStatus	reserved	DetailedStatusCode		16 (24)
ReplyPayload				20 (28)

Offset in () signifies offset for 64-bit context fields

Figure 4-2. Executive Class Reply Message Template

Fields

DetailedStatusCode	A more detailed description of the status when required. Values for this field depend on the Function. Chapter 3 provides the description of these codes for the Executive Class.
Function	The value from the Function field of the request.
InitiatorAddress	The value from the InitiatorAddress field of the request.
InitiatorContext	The value from the InitiatorContext field of the request.
MessageFlags	Set to indicate normal reply. Typically C0h (1100-0000b) for 32-bit context size, and C2h (1100-0010b) for 64-bit context size.
TargetAddress	The value from the TargetAddress field of the request.
ReqStatus	This field conveys the general status of the transaction as defined in Chapter 3.
ReplyPayload	Detailed information when required. Size and content of this field are defined by the particular function and status codes. Unless otherwise specified, the size of this field is zero for Executive class messages.
TransactionContext	The value from the TransactionContext field of the request.

4.4.1.2 Transport Failure Reply

When a message cannot be delivered or processed, a generic reply is returned with the FAIL bit set in the MessageFlags field. The format of this message is specified in chapter 3.

4.4.2 Utility Messages

All classes of I/O drivers must support a common group of utility messages. Not all utility messages apply to the Executive class, such as an ***UtilClaim*** message to an IOP. The IOP rejects inappropriate requests by a normal reply with an *UNSUPPORTED_FUNCTION* detailed status.

4.4.2.1 Get Parameters

The ***UtilParamsGet*** utility message specified in Chapter 6 reads a specified set of current operating parameters. The IOP copies that set of parameters into the buffer specified by the

SGL or places it in the ReplyPayload and then sends a reply. The parameter groups for the Executive class and their format are specified in Table 4-7. The normal reply to the ***UtilParamsGet*** request is specified in Chapter 6.

4.4.2.2 Set Parameters

The ***UtilParamsSet*** utility message specified in Chapter 6 causes the IOP to modify specified parameters. The parameter sets for the Executive class and their format are specified in Table 4-7. Only the host can send this message. The normal reply is specified in Chapter 6.

4.4.2.3 Event Registration

The host enables fault notification via the ***UtilEventRegister*** utility message (see Chapter 6). When the host enables events, the IOP sends the ***UtilEventRegister*** reply notifying the host when those events occur (e.g., the IOP detects that a peer IOP identified as operational in the last SysTab is no longer operational or is does not respond to requests).

The EventIndicator is a 32-bit enumerated value, specifying the source that triggered this event. Only one bit can be set and its location corresponds to the event category that triggered this event, as described by Table 4-4. (Also see common events specified in Chapter 6.)

Table 4-4. EventIndicator Assignments for Executive Class

Event Name	Bit	Description
RESOURCE_LIMITS	0	Resource limits exceeded (e.g., memory utilization)
CONNECTION_FAIL	1	Connection failure - detected that another IOP is not responding
ADAPTER_FAULT	2	Adapter fault - hardware assigned to IOP not present or not operational
POWER_FAIL	3	Power loss, running on auxiliary or standby power
RESET_PENDING	4	Reset pending (e.g., battery backup diminishing or NMI)
RESET_IMMINENT	5	Reset scheduled (e.g., parity error, protection violation, NMI, etc.)
HARDWARE_FAIL	6	Hardware failure, no recovery expected
XCT_CHANGE	7	Change to external connection table (i.e., new connection)
NEW_LCT_ENTRY	8	Entry added to the logical configuration table
MODIFIED_LCT	9	Logical configuration table entry modified
DDM_AVAILIBILITY	10	DDM failed or suspended

EventData in the ***UtilEventRegister*** reply message contains the information about the event. The structure of this field depends on the event category specified by the EventIndicator, as described in Table 4-5. (Also see common events specified in Chapter 6.)

Table 4-5. EventData For Executive Class Events

Event Name	EventData
ADAPTER_FAULT	HRT entry defining the adapter (Figure 4-20)
CONNECTION_FAIL	Bits 0-11 contain the IOP_ID of the failing IOP. Bits 12-15 contain one of the following codes: 0h = Responding normally 01h = Not responding – timed out waiting for response 02h = No available message frames – timed out trying to fetch an empty message frame. Bits 16-31 contain the failing IOP's HostUnitID
DDM_AVAILIBILITY	Bits 0-11 contain the TID of the failing DDM. Bits 12-15 contain one of the following codes: 0h = Responding normally 1h = Congested - event queue exceeded threshold 2h = Not responding - excessive execution time 3h = protection violation 4h = code violation
HARDWARE_FAIL	EventData is one of the following codes: 00h = unknown cause 01h = CPU failure 02h = memory fault 03h = DMA failure 04h = I/O bus failure
MODIFY_LCT	The modified logical configuration table entry.
NEW_LCT_ENTRY	The new logical configuration table entry.
POWER_FAIL	No EventData.
RESET_IMMINENT	EventData is one of the following values: 00h = unknown cause 01h = power loss 02h = code violation 03h = parity error 04h = code execution exception 05h = watchdog timer expired
RESET_PENDING	EventData is one of these values: 01h = power loss 02h = code violation
RESOURCE_LIMITS	A 32-bit field that contains an enumerated list of limited resources: Bit 0: Low memory Bit 1: Inbound message frame Free_List pool low Bit 2: Outbound message frame Free_List pool low
XCT_CHANGE	The new external connection table entry.

4.4.3 Executive Base Class Messages

4.4.3.1 Adapter Assign

The ***ExecAdapterAssign*** request assigns an adapter (or a physical device) to the specified HDM. The IOP adds the adapter to its HRT. The message indicates if the assignment is permanent (remembered across resets) or temporary (remembered until the next reset). If the host makes an assignment to a specific TID (not 000h), only the host can change it. Such an assign or release request by a DDM or another IOP must be rejected. The normal reply is a default with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecAdapterAssign	InitiatorAddress		TargetAddress=000h	4
InitiatorContext				8
TransactionContext				12 (16)
OperationFlags	reserved		DdmTID	16 (24)
AdapterAddress (HRT entry)				20 (28)

Offset in () signifies offset for 64-bit context fields

Figure 4-3. *ExecAdapterAssign* **Request Message**

Fields

AdapterAddress — Takes the form of an HRT entry. See Figure 4-20.

DdmTID — Indicates the module to which the adapter is assigned. A value of 000h assigns the adapter to the IOP, so the IOP can assign it according to its own configuration rules.

OperationFlags — Bit 0: Permanent. Values:
- 0 Assignment is temporary (until a reset).
- 1 Assignment is permanent.

Permanent assignments are saved in the IOP's permanent store and remembered over resets and power cycles.

4.4.3.2 Adapter Read

ExecAdapterRead requests that the IOP read the memory or registers of an adapter and place that information in the buffer specified by the SGL. Either the host or an IOP can send this message. The normal reply is a default reply with no reply payload.

For inventory management, this message provides the mechanism so a host-based resource manager can identify resources hidden behind an IOP. The primary purpose of the ***ExecAdapterRead*** message is reading any PCI configuration registers of a hidden PCI device. The generic format of this message supports other bus/adapter architectures as well.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset	0
ExecAdapterRead	InitiatorAddress		TargetAddress=000h	4
InitiatorContext				8
TransactionContext				12 (16)
AdapterID				16 (24)
RequestFlags				20 (28)
Offset				24 (32)
Length				28 (36)
SGL for result				32 (40)

Offset in () signifies offset for 64-bit context fields

Figure 4-4. *ExecAdapterRead* Request Message

Fields

AdapterID
: Value from HRT identifying the adapter. For PCI, each PCI function is listed with its own AdapterID.

Length
: Number of bytes to read. The IOP reads exactly the number of bytes indicated in the Length field.

Offset
: Address of registers or memory to read. This value is expressed as an offset, since the originator does not know the address where the adapter is configured. For PCI, the IOP treats multiple memory assignments as contiguous. For example, if there are two memory windows (1MB and 4MB), an offset of 3MB actually refers to the memory at location at 2MB offset in the second memory window.

Request Flags
: Bits 1::0 indicate address space as follows:
0 0 = Configuration Registers
0 1 = I/O Registers
1 0 = Adapter Memory

VersionOffset
: 81h for 32-bit context size and A1h for 64-bit context size.

4.4.3.3 Adapter Release

The ***ExecAdapterRelease*** message is the inverse of the ***ExecAdapterAssign*** message. The normal reply is a default reply with no payload. On receipt, the IOP issues a ***DdmAdapterRelease*** if the adapter had been attached to a DDM, releases all controls, such as interrupt steering, and then replies. A successful reply means that the host can load its own drivers for the adapter.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecAdapterRelease	InitiatorAddress		TargetAddress=000h	4
InitiatorContext				8
TransactionContext				12 (16)
OperationFlags	reserved			16 (24)
AdapterAddress (HRT entry)				20 (28)
				24 (32)

Offset in () signifies offset for 64-bit context fields

Figure 4-5. *ExecAdapterRelease* Request Message

Field

AdapterAddress Takes the form of an HRT entry. See Figure 4-20.

OperationFlags Bit 0: Permanent. Values:
0 Release is temporary (until a reset).
1 Release is permanent.
Permanent release deletes the configuration from the IOP's permanent store.

4.4.3.4 BIOS Information Set

This message identifies a device that was incorporated into the BIOS. When the BIOS provides access to an I_2O device such as a hard disk, the BIOS updates the BIOS Information field in the IOP's logical configuration table with this message. Only the BIOS can send this message. The IOP must preserve this information during the system transition from BIOS to OS, which might involve resetting the IOP (see ***ExecIopReset*** and ***ExecIopClear*** messages). The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecBiosInfoSet	InitiatorAddress		TargetAddress=000h	4
InitiatorContext				8
TransactionContext				12 (16)
BiosInfo	reserved		DeviceTID	16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 4-6. *ExecBiosInfoSet* Request Message

Field

BiosInfo The value specified in the BIOS function call that identifies the device. This value is placed in the BiosInfo field of the logical configuration table entry for the device identified by DeviceTID.

DeviceTID The TID of the device accessible via the BIOS.

4.4.3.5 Boot Device Set

This message indicates which device booted the OS. Only the BIOS can send this message. The IOP must preserve this information during the system transition from BIOS to OS which might involve resetting the IOP (***ExecIopReset*** or ***ExecIopClear***). The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecBootDeviceSet	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
reserved		BootDevice		16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 4-7. Boot *Device* Set Request Message

Field

BootDevice The TID of the device used to boot the OS. This value is placed in the BootDevice field of the logical configuration table.

4.4.3.6 Configuration Validate

This message indicates that the host accepts the current configuration as valid. The IOP changes the status of suspect drivers to current and may delete old drivers from its store. Only the host may send this message. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecConfigValidate	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)

Offset in () signifies offset for 64-bit context fields

Figure 4-8. *ExecConfigValidate* Request Message

4.4.3.7 Connection Setup

The ***ExecConnSetup*** message is used by an IOP to connect one of its DDMs and a device registered on another IOP. IOP1 and IOP2, respectively, refer to the IOPs sending and receiving the ***ExecConnSetup*** request message. The InitiatorDevice and TargetDevice, respectively, refer to the DDM on IOP1 that will send requests and the device on IOP2 that will receive them.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecConnSetup	InitiatorAddress	TargetAddress = 000h		4
InitiatorContext				8
TransactionContext				12 (16)
OperationFlags	InitiatorDevice	TargetDevice		16 (24)
reserved	IOP2AliasForInitiatorDevice	IOP1AliasForTargetDevice		20 (28)
reserved		IOP1InboundMFrameSize		24 (32)
MessageClass				28 (36)

Offset in () signifies offset for 64-bit context fields

Figure 4-9. *ExecConnSetup* Request Message

Fields

InitiatorAddress — Alias assigned to the requesting IOP (IOP1) by the target IOP (IOP2) in the ***ExecIopConnect*** transaction.

InitiatorDevice — The natural TID assigned by IOP1 to the device that will be sending request messages to the TargetDevice. When forwarding replies, IOP2 replaces the alias in the InitiatorAddress field with this value.

IOP1AliasForTargetDevice — IOP1's alias TID assigned to the TargetDevice. Its value uniquely identifies to IOP1 the IOP and device. When IOP1 forwards a request from the InitiatorDevice, it removes this value from the TargetAddress field replacing it with the natural TID assigned by IOP2. When IOP2 forwards the reply from the TargetDevice, IOP2 replaces the natural TID in the TargetAddress field with this value.

IOP1InboundMFrameSize — Number of 32-bit words in each message frame associated with the requester's inbound message queue.

IOP2AliasForInitiatorDevice — IOP2's alias TID assigned to the InitiatorDevice. If IOP1 previously connected with the InitiatorDevice, it reports the value previously assigned by the IOP2; otherwise, the value 000h denotes unknown.

MessageClass — The message class ID for requests being sent from the InitiatorDevice to the TargetDevice. See Chapter 6 for class code assignments.

OperationFlags — Bit 0 Values:

0 *CLIENT_SERVER* – supports only requests from InitiatorDevice to the TargetDevice.

1 *BIDIRECTIONAL* – supports requests in both directions.

Bits 1 - 7 are reserved

TargetAddress — 000h

TargetDevice — Identifies the subject of the request and contains the TID assigned to that device by the target IOP (IOP2). IOP1 replaces the value in the

TargetAddress field with this value when forwarding requests to that device.

Figure 4-10 shows the structure of the ***ExecConnSetup*** reply.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecConnSetup	InitiatorAddress = 000h		TargetAddress = alias	4
InitiatorContext				8
TransactionContext				12 (16)
ReplyStatusCode	InitiatorDevice		TargetDevice	16 (24)
reserved	IOP2AliasForInitiator		IOP1AliasForTarget	20 (28)
reserved		IOP2InboundMFrameSize		24 (32)

Offset in () signifies offset for 64-bit context fields

Figure 4-10. *ExecConnSetup* Reply Message

Fields

InitiatorAddress 000h.

InitiatorDevice Identifies the device that will be sending requests. This value comes directly from the ***ExecConnSetup*** request.

IOP1AliasForTargetDevice IOP1's alias TID assigned to the TargetDevice. This value is taken directly from the ***ExecConnSetup*** request.

IOP2AliasForInitiatorDevice Alias TID assigned to the InitiatorDevice by IOP2. If IOP2 previously connected to the InitiatorDevice, it uses the same value. Otherwise IOP2 assigns a value. Its value uniquely identifies to IOP2 the IOP and device. When IOP1 forwards a request from the InitiatorDevice, it removes the natural TID it assigned from the InitiatorAddress field and replaces it with this value. When IOP2 forwards the reply from the TargetDevice, IOP2 removes this value from the InitiatorAddress field and replaces it with the natural TID assigned by IOP1.

IOP2InboundMFrameSize

Number of 32-bit words in each message frame associated with the responder's inbound message queue.

TargetAddress IOP1's alias for IOP2, established by the **ExecIopConnect** transaction.

TargetDevice Identifies the subject of the request and contains the TID assigned to that device by IOP2. This value comes directly from the request.

4.4.3.8 DDM Destroy

ExecDdmDestroy removes a driver from operation. If the DdmTID is not registered as a DDM class device, this message is rejected. If the DdmTID is a DDM class device, then the IOP removes all instances (i.e., destroys all devices registered by the DDM). Only the host can send this message. The normal reply is a default reply with no payload. This message does not affect the presence of the DDM in the IOP's permanent store.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecDdmDestroy	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
reserved		DdmTID		16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 4-11. *ExecDdmDestroy* Request Message

Fields

DdmTID Indicates the module to destroy.

4.4.3.9 Ddm Enable

The ***ExecDdmEnable*** message is used during system configuration to notify the IOP that it can resume operations with a device on another IOP. Once the IOP replies to this message, it can send messages to the indicated TID. Only the host can send this message. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecDdmEnable	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
reserved		reserved	DeviceTID	16 (24)
HostUnitID		reserved	IOP_ID	20 (28)

Offset in () signifies offset for 64-bit context fields

Figure 4-12. ExecDdmEnable Request Message

Fields

DeviceTID Indicates the natural TID of the I/O device being enabled.

IOP_ID Indicates the IOP that contains the device being enabled.

HostUnitID Indicates the unit that contains the device being enabled

4.4.3.10 Ddm Quiesce

The ***ExecDdmQuiesce*** message is used during system configuration to notify the IOP that a device on another IOP is being reconfigured. The IOP must stop sending messages to the indicated device. Once the IOP replies to this message, it does not send messages to the indicated TID until it receives an ***ExecDdmEnable*** or ***ExecSysEnable*** message. Only the host can send this message. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecDdmQuiesce	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
reserved		reserved	DeviceTID	16 (24)
HostUnitID		reserved	IOP_ID	20 (28)

Offset in () signifies offset for 64-bit context fields

Figure 4-13. *ExecDdmQuiesce* Request Message

Fields

DeviceTID — Indicates the natural TID of the I/O device being quiesced.

IOP_ID — Indicates the IOP that contains the device being quiesced.

HostUnitID — Indicates the unit that contains the device being quiesced

4.4.3.11 Ddm Reset

The ***ExecDdmReset*** message is used during system configuration to notify the IOP that a device on another IOP has been reinitialized. The IOP must stop sending messages to the indicated device and delete any connections to it. Once the IOP replies to this message, it does not send messages to the indicated TID until it receives an ***ExecDdmEnable*** or ***ExecSysEnable*** message. Only the host can send this message. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecDdmReset	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
reserved		reserved	DeviceTID	16 (24)
HostUnitID		reserved	IOP_ID	20 (28)

Offset in () signifies offset for 64-bit context fields

Figure 4-14. *ExecDdmReset* Request Message

Fields

DeviceTID — Indicates the natural TID of the I/O device being reset.

IOP_ID — Indicates the IOP that contains the device being reset.

HostUnitID — Indicates the unit that contains the device being reset.

4.4.3.12 Ddm Suspend

ExecDdmSuspend suspends operation of a driver. If the DdmTID is not registered as a DDM class device, then the IOP suspends that instance of the driver. If the DdmTID is the TID assigned to a DDM, the message suspends all instances (all devices registered by that DDM). Only the host can send this message. The normal reply is a default reply with no payload.

<table>
<tr><th>31 3 24</th><th>23 2 16</th><th colspan="2">15 1 8</th><th>7 0 0</th><th>offset</th></tr>
<tr><td colspan="2">MessageSize</td><td colspan="2">MessageFlags</td><td>VersionOffset = 01h</td><td>0</td></tr>
<tr><td><b><i>ExecDdmSuspend</i></b></td><td colspan="2">InitiatorAddress</td><td colspan="2">TargetAddress=000h</td><td>4</td></tr>
<tr><td colspan="5">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="5">TransactionContext</td><td>12 (16)</td></tr>
<tr><td colspan="3">reserved</td><td colspan="2">DdmTID</td><td>16 (24)</td></tr>
</table>

Offset in () signifies offset for 64-bit context fields

Figure 4-15. *ExecDdmSuspend* Request Message

4.4.3.13 Device Assign

The ***ExecDeviceAssign*** request assigns an I/O device registered by one DDM to the specified ISM. The device can be on the same or a different IOP than the designated module. If the host assigns a device to a specific DdmTID (not 000h), then only the host can change that assignment. An assign or release request by a DDM or another IOP must be rejected. The normal reply is a default reply with no payload.

<table>
<tr><th>31 3 24</th><th>23 2 16</th><th colspan="2">15 1 8</th><th>7 0 0</th><th>offset</th></tr>
<tr><td colspan="2">MessageSize</td><td colspan="2">MessageFlags</td><td>VersionOffset = 01h</td><td>0</td></tr>
<tr><td><b><i>ExecDeviceAssign</i></b></td><td colspan="2">InitiatorAddress</td><td colspan="2">TargetAddress=000h</td><td>4</td></tr>
<tr><td colspan="5">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="5">TransactionContext</td><td>12 (16)</td></tr>
<tr><td>OperationFlags</td><td colspan="2">DdmTID</td><td colspan="2">DeviceTID</td><td>16 (24)</td></tr>
<tr><td colspan="2">HostUnitID</td><td>reserved</td><td colspan="2">IOP_ID</td><td>20 (28)</td></tr>
</table>

Offset in () signifies offset for 64-bit context fields

Figure 4-16. *ExecDeviceAssign* Request Message

Fields

DdmTID — Indicates the module to which the I/O device is assigned.

DeviceTID — Indicates the natural TID of the I/O device being assigned.

HostUnitID — Indicates the unit that contains the device being assigned.

IOP_ID — Indicates the IOP that contains the device being assigned.

OperationFlags — Bit 0: Permanent. Values:
0 Assignment is temporary (until a reset).
1 Assignment is permanent.
Permanent assignments are saved in the IOP's permanent store and remembered over resets and power cycles.

4.4.3.14 Device Release

The ***ExecDeviceRelease*** message is the inverse of the ***ExecDeviceAssign*** message. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecDeviceRelease	InitiatorAddress		TargetAddress=000h	4
InitiatorContext				8
TransactionContext				12 (16)
OperationFlags	DdmTID		DeviceTID	16 (24)
HostUnitID		reserved	IOP_ID	20 (28)

Offset in () signifies offset for 64-bit context fields

Figure 4-17. *ExecDeviceRelease* Request Message

Field

DdmTID Indicates the module to which the I/O device is assigned.

DeviceTID Indicates the natural TID of the I/O device being released.

HostUnitID Indicates the unit that contains the device being released.

IOP_ID Indicates the IOP that contains the device being released.

OperationFlags Bit 0: Permanent. Values:
0 Release is temporary (until a reset).
1 Release is permanent.
Permanent release deletes the configuration from the IOP's permanent store.

4.4.3.15 HRT Get

The IOP also provides a hardware resource table, which tells the host (and other IOPs) of any adapters controlled by the IOP. In general, the HRT lists all adapters and locations that the IOP can control.

The host or another IOP obtains a copy of the IOP's hardware resource table by sending the ***ExecHrtGet*** message. The only parameter is a scatter-gather list containing a single buffer in which the table is placed.

If the buffer is too small for the entire table, the IOP copies as much of the table as can fit. Since the beginning of the table describes its size, the initiator can resubmit with a larger buffer. The normal reply is a default reply with no payload.

The host must detect and resolve the conflict when two IOPs indicate control of the same adapter.

This function is crucial for assuring system integrity during initialization and must be simple enough for the BIOS and OS to configure around the IOP. BIOS functionality must remain simple and the OS functionality occurs before the OS loads I_2O drivers. In addition, all changes to this message and the HRT structure must provide backward compatibility.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset	0
ExecHrtGet	InitiatorAddress		TargetAddress=000h	4
InitiatorContext				8
TransactionContext				12 (16)
SGL				16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 4-18. *ExecHrtGet* Request Message

VersionOffset 41h for 32-bit context size and 61h for 64-bit context size.

SGL Identifies the buffer where the IOP copies the HRT.

31 3 24	23 2 16	15 1 8	7 0 0	offset
HrtVersion	EntryLength	NumberEntries		0
CurrentChangeIndicator				4
HrtEntry 1 (see Figure 4-20)				8
HrtEntry 2 (see Figure 4-20)				
...				
HrtEntry *n* (see Figure 4-20)				

Figure 4-19. Hardware Resource Table Structure

Fields

CurrentChangeIndicator Initialized to 0 and conditionally incremented on receiving a table read request. Incremented only if the table changed since the last read request.

EntryLength Length of each entry in 32-bit words. Maximum value of 0FFh is 1020 bytes.

Hrt Entry See Figure 4-20.

HrtVersion 00h for this version of the document.

NumberEntries Number of entries contained in the table.

Each HRT entry has the following format:

31 3 24	23 2 16	15 1 8	7 0 0	offset
AdapterID				0
BusType	BusNumber	AdapterState	Controlling TID	4
PhysicalLocation				8

Figure 4-20. HRT Entry

Fields for HRT Entry

AdapterID — Arbitrary value assigned to the adapter by the IOP used to identify the adapter in other messages.

AdapterState — State of the adapter:

Value	*Assigned*	*Present*	*Controlled*	*Hidden*
0	no	n/a	no	no
1	reserved			
2	yes	no	n/a	no
3	yes	no	n/a	yes
4	yes	yes	no	no
5	yes	yes	no	yes
6	yes	yes	yes	no
7	yes	yes	yes	yes
8	reserved			
9	reserved			
10	HARD	no	n/a	no
11	HARD	no	n/a	yes
12	HARD	yes	no	no
13	HARD	yes	no	yes
14	HARD	yes	yes	no
15	HARD	yes	yes	yes

ASSIGNED means that the slot or location is assigned to the IOP and if an adapter is detected, the IOP assigns its control to a DDM. Assignment is granted and revoked via the ***ExecAdapterAssign*** and ***ExecAdapterRelease*** messages. An adapter or location listed in this table must be assignable to the IOP. The value of ***HARD*** indicates that only the IOP can control the adapter or location (e.g., an embedded controller) and it cannot be assigned to the host or another IOP.

PRESENT means that the IOP detected an adapter in the specified location.

CONTROLLED means that the adapter is attached to a DDM and controlled by the IOP. Reporting an adapter that is *PRESENT* but not *CONTROLLED* indicates the IOP lacks a suitable driver for that adapter.

HIDDEN means that the configuration space of that location or adapter is not visible to the host.

BusNumber Arbitrary value the IOP assigns to identify the physical bus where the adapter resides. Adapters residing on the same physical bus should report the same BusNumber.

BusType Indicates the type of expansion bus. For values, see *Common Structures for Adapters* in Chapter 3.

Controlling TID Local TID of the HDM to which this adapter is assigned. If the AdapterState field indicates the adapter is not assigned to the IOP, this value is set to 001h. If the adapter is assigned to the IOP but not controlled, this value is set to FFFh. A value of 000h means the IOP itself controls the adapter and does not intend to assign it to a DDM.

PhysicalLocation Eight bytes of data that identify the adapter, physical device, or function. Format of this field depends on BusType, and those variations are defined in *Common Structures for Adapters* in Chapter 3.

4.4.3.16 IOP Clear

The ***ExecIopClear*** message causes the IOP to terminate external operations, clear all of its input queues and prepare for a system restart. Use this command when rebooting the system or when the system reconfigures (for example, changing the system addresses of IOPs and adapters). Internal operation of the IOP continues normally. Only the host can send this message. Unlike the ***ExecIopReset***, this message does not reset the outbound queue and thus the IOP can reply to this message. The normal reply is a default reply with no payload.

The IOP is not expected to rebuild its LCT as a result of this message. If it does, it must preserve the LCT's BootDevice and BiosInfo fields so they are available, unchanged, after the IOP initializes. This ensures a smooth transition from BIOS to OS.

<table>
<tr><th colspan="2">31 3 24</th><th colspan="2">23 2 16</th><th colspan="2">15 1 8</th><th colspan="2">7 0 0</th><th>offset</th></tr>
<tr><td colspan="4">MessageSize</td><td colspan="2">MessageFlags</td><td colspan="2">VersionOffset = 01h</td><td>0</td></tr>
<tr><td colspan="2"><b><i>ExecIopClear</i></b></td><td colspan="3">InitiatorAddress</td><td colspan="3">TargetAddress=000h</td><td>4</td></tr>
<tr><td colspan="8">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="8">TransactionContext</td><td>12 (16)</td></tr>
</table>

Offset in () signifies offset for 64-bit context fields

Figure 4-21. *ExecIopClear* Request Message

The host is expected to send a ***ExecSysQuiesce*** message to all IOPs before sending a ***ExecIopClear*** message. When it receives this message, the IOP suspends external message service, flushes its inbound message queue and all path information, and reallocates its primary inbound queue. Once the IOP curtails operation and rebuilds its inbound queues, it replies to the message. The IOP sends ***DdmPathReset*** to all DDMs for each alias TID (from an external connection). This includes TID 001h. Consequently, the UserTID field in the LCT for external users gets reset to *not claimed*.

4.4.3.17 IOP Connect

An IOP uses the ***ExecIopConnect*** message to set up a path to another IOP. This path is used to exchange Executive class messages between the IOPs. This is necessary to set up peer connections.

The IOP1 and IOP2, respectively, refer to the entities sending and receiving the ***ExecIopConnect*** request.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecIopConnect	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
IOP1MsgerType	reserved	reserved		16 (24)
reserved	IOP1AliasForIOP2	IOP1InboundMFrameSize		20 (28)
HostUnitID1		reserved	IOP_ID1	24 (32)

Offset in () signifies offset for 64-bit context fields

Figure 4-22. *ExecIopConnect* Request Message

Fields

HostUnitID1 — HostUnitID for IOP1.

InitiatorAddress — Initiator address is always 000h, because the alias for IOP1 is not yet established.

IOP1AliasForIOP2 — Originator's alias TID for IOP2. IOP2 places this value in the InitiatorAddress field when sending request messages to IOP1 (the initiator of this request), and in the TargetAddress field when replying to requests from IOP1. This tells IOP2 which IOP sourced the message.

IOP_ID1 — IOP_ID assigned to IOP1 by the host. The value FFFh denotes value not assigned or not known.

IOP1MsgerType — IOP1's messenger type. The only type defined by this version of the specification is: 00h = memory mapped message unit.

IOP1InboundMFrameSize — Number of 32-bit words in each message frame associated with IOP1's inbound message queue.

TargetAddress — Target address is always 000h.

When the status is success, the structure of the ***ExecIopConnect*** reply is:

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecIopConnect	InitiatorAddress = 000h	TargetAddress = alias		4
InitiatorContext				8
TransactionContext				12 (16)
ReplyStatusCode		DetailedStatusCode		16 (24)
reserved	IOP2AliasForIOP1	IOP2InboundMFrameSize		20 (28)
HostUnitID2		reserved	IOP_ID2	24 (32)

Offset in () signifies offset for 64-bit context fields

Figure 4-23. ExecIopConnect Reply Message

Fields

HostUnitID2 HostUnitID for IOP2.

InitiatorAddress 000h

IOP2AliasForIOP1 IOP2's alias TID for IOP1. IOP1 places this value in the InitiatorAddress field when sending request messages to IOP2, and in the TargetAddress field when replying to IOP2. This tells IOP2 which IOP sourced the message.

IOP2InboundMFrameSize
Number of 32-bit words in each message frame associated with IOP2's inbound message queue.

IOP_ID2 The IOP_ID assigned to IOP2 by the host. The value FFFh denotes value not assigned.

TargetAddress IOP1's alias for IOP2. This is the value from the IOP1AliasForIOP2 field of the request.

4.4.3.18 IOP Reset

When the IOP receives the ***ExecIopReset*** message, it terminates external operations: It clears its input and output queues, terminates all DDMs, and reloads the IOP's operating environment and all local DDMs. The host uses this command for recovering an ill IOP or completely reconfiguring the system. Only the host can send this message.

When it receives this message, the IOP rebuilds its LCT. The IOP must preserve the LCT's BootDevice and BiosInfo fields, so they are available, unchanged, after the IOP initializes. This ensures a smooth transition from BIOS to OS.

Since the IOP loses its state, a normal reply is not appropriate. Therefore, the IOP acknowledges the message by writing a status word to the location specified in the request before it attempts its operation. The IOP is not required to remember this address across the RESET; it writes only an acknowledge status, never a completion status. Rather, the host determines the status of the RESET by sending ***ExecStatusGet*** requests.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecIopReset	InitiatorAddress	TargetAddress=000h		4
Reserved (Universal Context Area) 16 bytes				8
StatusWordLowAddress				24
StatusWordHighAddress				28

Offset in () signifies offset for 64-bit context fields

Figure 4-24. *ExecIopReset* Request Message

This message reserves an area for universal context that may be used for tracing or debugging. The universal context field's size accommodates both a 64-bit InitiatorContext and TransactionContext. The host may place any value in this field. Since there is no reply, the IOP ignores this field. Also the status word address is a 64-bit value. If the StatusWordHighAddress does not contain zero, then the IOP may ignore the request.

The structure of the status word written to the reply address is specified in Figure 4-25.

31 3 24	23 2 16	15 1 8	7 0 0	
reserved			ResetStatus	0

Figure 4-25. *ExecIopReset* Status Word Structure

Field

ResetStatus	Status of reset request. Values: 01h *IN_PROGRESS* 02h *REJECTED*

When it receives this message, the IOP:

1. Stops posting messages to its outbound queue.
2. Recovers any outstanding message frames in use internally.
3. Recovers all MFAs from the inbound free list.
4. Recovers all MFAs from the inbound post list.
5. Clears the inbound free list and post list. This must be done before it writes the StatusWord.
6. Writes *IN_PROGRESS* to the StatusWord only after it has cleared its inbound queues. The IOP's inbound free list can remain empty until the IOP boots. Writing the StatusWord only after the inbound free list has been cleared assures the host that once it detects that the IOP has written the StatusWord, it can safely fetch an inbound message frame and thus post an ***ExecStatusGet*** message. There is no guarantee that an inbound message frame will be available.

After the IOP writes the StatusWord, the host polls the free list until a message frame is available. Then, the host can immediately post a message and the IOP must be able to process it.

The host must send an ***ExecSysQuiesce*** message to all IOPs before sending an ***ExecIopReset*** message. The IOP clears its inbound message queues, writes the StatusWord, and jumps to its boot code. The boot code creates new inbound message frames and posts them to the free list. The host can now send the ***ExecStatusGet*** message.

4.4.3.19 LCT Notify

ExecLctNotify requests that an IOP place a copy of its logical configuration table in the buffer specified by the SGL. This happens only when the IOP's current Change Indicator is a different value than the LastReportedChangeIndicator field. Either the host or an IOP can send this message to other IOPs. DDMs may send this message to the local IOP to ascertain when configuration is complete. The normal reply is a default reply with no payload. Also, see the ***UtilEventRegister*** utility message specified in Chapter 6.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset	0
ExecLctNotify	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
ClassIdentifier				16 (24)
LastReportedChangeIndicator				20 (28)
SGL for LCT				24 (32)

Offset in () signifies offset for 64-bit context fields

Figure 4-26. *ExecLctNotify* Request Message

Fields

ClassIdentifier	Specifies the class of devices to include in the reply (FFFF-FFFFh = all classes). The IOP returns all logical configuration table entries that match ClassIdentifier, including those that have not changed.
LastReportedChangeIndicator	The reply occurs immediately if any table entries for the specified class have a Change Indicator value greater than this field. Otherwise, the reply is deferred until such a change occurs. A value of 0000-0000h assures an immediate reply.
SGL for LCT	Specifies a buffer to hold the logical configuration table. If the buffer is too small for the entire table, the IOP copies as much as fits. Since the beginning of the table describes its size, the initiator can resubmit with a larger buffer.
VersionOffset	61h for 32-bit context size and 81h for 64-bit context size.

31 3 24	23 2 16	15 1 8	7 0 0	offset
LctVer	BootDevice	TableSize		0
IopFlags				4
CurrentChangeIndicator				8
LctEntry 1 (as per Chapter 3)				12
LctEntry 2 (as per Chapter 3)				
...				
LctEntry *n* (as per Chapter 3)				

Figure 4-27. Logical Configuration Table Structure

Fields

BootDevice — TID of the device that booted the OS. Set to zero if none or unknown. The BIOS sets this value via the ***ExecBootDeviceSet*** message.

CurrentChangeIndicator — Initialized to 0 and incremented before the table is returned, only if the table changed since the last table read response.

IopFlags — Bit 0: Set to indicate that the IOP requests a configuration dialogue. All other bits reserved.

LctEntry — The content of a logical configuration table entry is defined in Chapter 3.

LctVer — Table version (0000b for this version).

TableSize — Number of 32-bit words in the table including this field.

4.4.3.20 Outbound Initialize

The ***ExecOutboundInit*** message causes the IOP to flush its outbound message queue (i.e., empty both FIFOs). Only the host can send this message. There is no reply message to this request since the outbound Free_List will be empty. The IOP conveys status by writing a status word to the system address specified in the SGL.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset	0
ExecOutboundInit	InitiatorAddress	TargetAddress = 000h		4
InitiatorContext				8
TransactionContext				12 (16)
HostPageFrameSize				16 (24)
OutboundMFrameSize		reserved	InitCode	20 (28)
SGL				24 (32)

Offset in () signifies offset for 64-bit context fields

Figure 4-28. *ExecOutboundInit* Request Message

Field

HostPageFrameSize Size of host’s page frame specified in number of bytes.

InitCode The host provides this code point and the IOP reports it in each ***ExecStatusGet*** response. Its value may also be modified by the ***UtilParamsSet*** message. The use of this field helps synchronize the initialization of the IOP between BIOS, BIOS extensions, ROM BIOS, and the OS. This field allows those entities not only to identify who is in control, but also the phase of initialization. To this end, ranges are assigned to each major entity. Each can set any value within its assigned range (see Exec Parameter Group 0001h).

OutboundMFrameSize Size of message frames for the outbound message queue specified in number of 32-bit words.

SGL Specifies one or two buffers. The first buffer must be exactly four bytes long and provide the location where the IOP writes the status word. The host supplies a second buffer if it wants the IOP to provide a list of MFAs of outbound message frames from the previous session. This allows the host to recover those resources.

VersionOffset 61h for 32-bit context size and 81h for 64-bit context size.

Since executing this message resets the outbound message queue, there is no reply to it. The IOP indicates its progress by writing a status word to the system location specified by the SGL. The IOP should initially report the status of *IN_PROGRESS* to acknowledge the request and then subsequently update the status word to indicate completion.

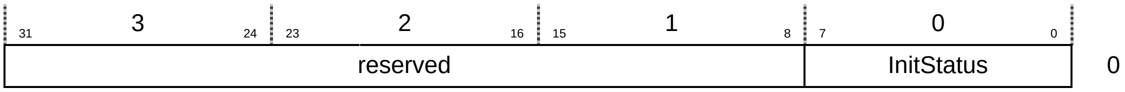


Figure 4-29. *ExecOutboundInit* Status Word Structure

Fields for Status Word

InitStatus Values:

00h Never used - The host may initialize to this value to detect when the InitStatus has been written.

01h *IN_PROGRESS*

02h *REJECTED*

03h *FAILED*

04h *COMPLETE*

When the IOP receives this message, it:

1. Writes *IN_PROGRESS* to the InitStatus word.
2. Stops posting messages to its outbound queue.
3. Recovers any outstanding message frames in use internally.
4. Removes all outbound message frame MFAs from the outbound free list.
5. Removes all outbound message frame MFAs from the outbound post list.

6. Builds a list (Figure 4-30) of all MFAs, from steps 2, 3, and 4 above, and places that list in the buffer specified by the SGL. If the list exceeds the buffer, the IOP places as much of it as fits. If the host does not want the list, it simply gives an SGL a single IGNORE element (e.g., the value C0-00-00-01h).
7. Writes *COMPLETE* to the InitStatus word.

31 3 24	23 2 16	15 1 8	7 0 0	offset
Count of MFAs in this list				0
Count of MFAs released				4
MFA				8
...				
MFA				

Figure 4-30. Reclaim List Structure

4.4.3.21 Path Enable

Receiving the ***ExecPathEnable*** message during system configuration notifies the IOP that it can resume operations with another IOP. Once the IOP replies, it can allocate and send messages to the indicated IOP. Only the host can send this message. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecPathEnable	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
HostUnitID		reserved	IOP_ID	16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 4-31. *ExecPathEnable* Request Message

4.4.3.22 Path Quiesce

Receiving the ***ExecPathQuiesce*** message during system configuration notifies the IOP that another IOP is being reconfigured. The IOP must stop sending messages to the indicated IOP. Once the IOP replies to this message, it does not allocate or send messages to the indicated IOP until it receives an ***ExecPathEnable*** or ***ExecSysEnable*** message. Only the host can send this message. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecPathQuiesce	InitiatorAddress		TargetAddress=000h	4
InitiatorContext				8
TransactionContext				12 (16)
HostUnitID		reserved	IOP_ID	16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 4-32. *ExecPathQuiesce* Request Message

4.4.3.23 Path Reset

Receiving the ***ExecPathReset*** message during system configuration notifies the IOP that another IOP has been reset. The IOP must stop sending messages to the indicated IOP, discard any outstanding message frames, and delete any connections with the other IOP. Once the IOP replies to this message, it does not allocate or send messages to the indicated IOP until it receives an ***ExecPathEnable*** or ***ExecSysEnable*** message. Only the host can send this message. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecPathReset	InitiatorAddress		TargetAddress = 000h	4
InitiatorContext				8
TransactionContext				12 (16)
HostUnitID		reserved	IOP_ID	16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 4-33. *ExecPathReset* Request Message

4.4.3.24 Static Message Frame Create

Because numerous messages routinely carry the exact same header and payload, there is a need for static (permanent) messages. A static message is a message frame reserved for one particular message. The sending entity fills in the message frame once. Then, each time it sends the message, only the address of the message frame (i.e., its MFA) posts to the target's inbound message FIFO. The message itself does not need to be copied across the system I/O bus.

Static messages that are sent repeatedly from one module to another without a change to the header or payload can do so very efficiently. An example is a notification message that an ISM sends to a peer identifying new data in an input pipe. By keeping a permanent copy of this message at the destination, the sourcing IOP does not copy the message each time it is delivered. As a further optimization, the receiving IRTOS may create a static event block for that message so it does not parse through the TID's tables each time it receives the message.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecStaticMfCreate	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
reserved			MaxOutstanding	16 (24)
StaticMessageFrame (Header + Payload)				20 (28)

Offset in () signifies offset for 64-bit context fields

Figure 4-34. *ExecStaticMfCreate* Request Message

Fields

MaxOutstanding — The maximum number of outstanding instances of this message frame. The initiator must not send the message so often that it appears more than MaxOutstanding times in the target IOP's inbound message queue. This ensures that the IOP's inbound queue can always accept a message frame. The IOP rejects requests that exceed the inbound message FIFO size.

StaticMessageFrame — The remainder of the message is the static message frame, which includes the message header and payload. The target IOP builds a static message frame from this field.

Figure 4-35 shows the structure of the ***ExecStaticMfCreate*** reply.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecStaticMfCreate	InitiatorAddress	TargetAddress = 000h		4
InitiatorContext				8
TransactionContext				12 (16)
ReqStatus	reserved	DetailedStatusCode		16 (24)
StaticMFA				20 (28)

Offset in () signifies offset for 64-bit context fields

Figure 4-35. *ExecStaticMfCreate* Reply Message

Fields

MessageFlags — Typically, C0h for 32-bit context size and C2h for 64-bit context.

StaticMFA — The MFA the initiator posts to the target's inbound message queue to send the static message.

4.4.3.25 Static Message Frame Release

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecStaticMfRelease	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
StaticMFA				16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 4-36. *ExecStaticMfRelease* Request Message

Fields

StaticMFA — The MFA that was returned in the ***ExecStaticMfCreate*** reply message.

The ***ExecStaticMfRelease*** reply is a standard reply with no payload.

4.4.3.26 Status Get

The ***ExecStatusGet*** message causes the IOP to place its status block in the buffer specified by the system address indicated in the ReplyBufferAddress field. This message can be sent by the host or an IOP. There is no reply to this request, which allows sending this request before the IOP's outbound queue is initialized or the IOP's state is known.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		Flags	VersionOffset = 01h	0
ExecStatusGet	InitiatorAddress	TargetAddress=000h		4
Reserved (UniversalContext) 16 bytes				8
ReplyBufferAddressLow				24
ReplyBufferAddressHigh (reserved)				28
ReplyBufferLength				28

Offset in () signifies offset for 64-bit context fields

Figure 4-37. *ExecStatusGet* Request Message

Fields

MessageFlags — Typically 00h for 32-bit context size and 02h for 64-bit context size

ReplyBufferAddress — Defined as a 64-bit field to accommodate requests from both 32-bit and 64-bit systems. If the ReplyBufferAddressHigh field does not contain zero, then the IOP may ignore the request (only 32-bit operation is defined).

ReplyBufferLength — Size of reply buffer in bytes. The IOP copies no more than this number bytes of the reply structure to the reply buffer starting with the first word. This field allows the initiator to request only the portion of the status structure it needs or expects. Thus, it provides compatibility when future versions of the specification append information to the reply structure.

UniversalContext This message has an area reserved for universal context that may be used for tracing or debugging purposes. The OS may place any value in this field. Since there is no reply, the IOP ignores this field.

Figure 4-38 depicts the status block that the IOP places in the buffer. It uses the location indicated by the system address in the ReplyBufferAddress field.

<table>
<tr><th>31 3 24</th><th>23 2 16</th><th>15 1 8</th><th>7 0 0</th><th>offset</th></tr>
<tr><td colspan="2">reserved</td><td colspan="2">OrganizationID</td><td>0</td></tr>
<tr><td colspan="2">HostUnitID</td><td>reserved</td><td>IOP_ID</td><td>4</td></tr>
<tr><td>MessengerType</td><td>IopState</td><td>I2oVersion</td><td>SegmentNumber</td><td>8</td></tr>
<tr><td>reserved</td><td>InitCode</td><td colspan="2">InboundMFrameSize</td><td>12</td></tr>
<tr><td colspan="4">MaxInboundMFrames</td><td>16</td></tr>
<tr><td colspan="4">CurrentInboundMFrames</td><td>20</td></tr>
<tr><td colspan="4">MaxOutboundMFrames</td><td>24</td></tr>
<tr><td colspan="4">ProductIDString</td><td>28
48</td></tr>
<tr><td colspan="4">ExpectedLctSize</td><td>52</td></tr>
<tr><td colspan="4">IopCapabilities</td><td>56</td></tr>
<tr><td colspan="4">DesiredPrivateMemSize</td><td>60</td></tr>
<tr><td colspan="4">CurrentPrivateMemSize</td><td>64</td></tr>
<tr><td colspan="4">CurrentPrivateMemBase</td><td>68</td></tr>
<tr><td colspan="4">DesiredPrivateIOSize</td><td>72</td></tr>
<tr><td colspan="4">CurrentPrivateIOSize</td><td>76</td></tr>
<tr><td colspan="4">CurrentPrivateIOBase</td><td>80</td></tr>
<tr><td>FFh</td><td colspan="3">reserved</td><td>84</td></tr>
</table>

Figure 4-38. Status Block Structure

Fields

CurrentInboundMFrames Current number of message frames created for the inbound message frame pool.

CurrentPrivateIOBase The base address of the Private I/O space currently allocated to this IOP. This value is set to zero at power up.

CurrentPrivateIOSize Number of bytes of Private I/O space currently allocated to this IOP. This value is set to zero at power up.

CurrentPrivateMemBase The base address of the Private Memory space currently allocated to this IOP. This value is set to zero at power up.

CurrentPrivateMemSize Number of bytes of Private Memory space currently allocated to this IOP. This value is zero when the IOP is powered on.

DesiredPrivateIOSize — The number of bytes of Private I/O space requested by the IOP.

DesiredPrivateMemSize — The number of bytes of Private Memory space requested by the IOP.

ExpectedLctSize — The total expected size (number of bytes) of the IOP's logical configuration table. This estimate is based on the number of TIDs assigned, even if table contains no entry for that TID.

HostUnitID — Value from ***ExecSysTabSet*** message

I2oVersion — Version of I$_2$O specification under which the IOP is operating. This version = 01h.

InboundMFrameSize — Size of the inbound message frame (in 32-bit words). Minimum allowable size is 16 (i.e., 64 bytes).

InitCode — Initialization code. The IOP initially sets InitCode to 00h and the host updates the value indicating the progress of the initialization sequence. See Table 4-7.

IOP_ID — Value from ***ExecSysTabSet*** message. Arbitrary number the host resource manager assigns to uniquely identify each IOP. If a value is not yet assigned, then the IOP reports the value FFFh.

IopCapabilities: — Bit-specific fields that indicate the IOP's current modes and capabilities.

Bits 1,0; ContextFieldSizeCapability

- 0,0 Supports only 32-bit context fields.
- 0,1 Supports only 64-bit context fields.
- 1,0 Supports 32-bit & 64-bit context fields, but not concurrently.
- 1,1 Supports 32-bit & 64-bit context fields concurrently.

Bits 3,2; CurrentContextFieldSize

- 0,0 not configured.
- 0,1 Supports 32-bit context fields only.
- 1,0 Supports 64-bit context fields only.
- 1,1 Supports both 32-bit or 64-bit context fields concurrently.

Bit 4: InboundPeerSupport - This IOP supports connections from ISMs on other IOPs to devices registered on this IOP (peer ISM can Claim local device)

Bit 5: OutboundPeerSupport - This IOP supports connections from ISMs on this IOP to devices registered on other IOPs.

Bit 6: PeerToPeerSupport - This IOP supports peer-to-peer connections via Peer Transport Protocol (tbd).

Other bits reserved.

IopState — Values (see Table 4-1 for state definition):

01h *INIT*

02h	*RESET*
04h	*HOLD*
05h	*READY*
08h	*OP*
10h	*FAILED*
11h	*FAULTED*

MaxInboundMFrames — Maximum number of message frames that can be allocated for the inbound message queue.

MaxOutboundMFrames — Number of message frames that can be allocated for the outbound message queue.

MessengerType — The only type defined by this version of the specification is:
00h = memory mapped message unit.

OrganizationID — ID assigned by I_2O SIG to the vendor of the IOP.

ProductIDString — 24 bytes of ASCII text identifying the product. It can be any value the vendor supplies. The value identifies the product for troubleshooting, upgrading, and so forth.

SegmentNumber — Value from ***ExecSysTabSet*** message. Set to zero if not established.

The last byte of the table is fixed at 0FFh, so the initiator has a positive indication of when all the information is written to the status block.

4.4.3.27 Software Download

The host transfers new software to the IOP using the ***ExecSwDownload*** message. The software to download may be one of the following:

- DDM
- DDM Module Parameter Block
- DDM Configuration Dialogue Table
- IRTOS Upgrade
- IRTOS Private Module
- IRTOS Configuration Dialogue Table
- IOP Private Module
- IOP Configuration Dialogue Table

This message instructs the IOP to either install the software module into its permanent store, or load it into memory for execution. If the installed software is a DDM, the IOP determines if it loads and initializes the DDM: It matches the DDM to a device or an adapter assigned to the IOP that is not otherwise controlled. If the installed module is an IRTOS upgrade, the IOP loads the new IRTOS during the next boot. The behavior of the IOP for other forms of software download is implementation dependent. Only the host can send this message. The normal reply is a default reply with no payload.

The host can break a single software module into fragments, conveying each fragment with an individual ***ExecSwDownload*** message. All messages must have the same transaction context.

The IOP must recombine the fragments into a single module. The IOP uses the transaction context to ensure that all ***ExecSwDownload*** messages are part of the same sequence.

The host must download only one fragment of the sequence at a time, wait for the reply, and then send the next fragment of the same module. Only one sequence may be in progress at a time. Interleaving sequences causes an error.

The host must send all fragments in order. The first fragment of the sequence (Current Fragment = 1) contains the beginning of the software module; the last fragment (Current Fragment = Total Fragments) contains the last portion of the software module. The IOP aborts any previous download if a request arrives specifying the first fragment (Current Fragment = 1) of a new sequence. After the first fragment, if any fragment arrives out of order, the IOP returns an error code in the reply and aborts the operation.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset	0
ExecSwDownload	InitiatorAddress = 001h		TargetAddress = 000h	4
InitiatorContext				8
TransactionContext				12 (16)
DownloadFlags	SwType	TotalFragments	CurrentFragment	16 (24)
SWSize				20 (28)
SwID				24 (32)
SGL :				28 (36)

Offset in () signifies offset for 64-bit context fields

Figure 4-39. *ExecSwDownload* Request Message

Fields

CurrentFragment Value indicating the position of the fragment in the sequence. The first fragment is 1, the second fragment is 2, and so forth.

DownloadFlags Describes the download operation:

Bit	Name	Description
0	DownloadType	0 = Load: put module in memory 1 = Install: put module in the IOP's permanent store
1	SafetyOverride	0 = The mode is tagged experimental and the previous version tagged "old". 1 = The module is tagged verified, deleting any previous version.
2-7		reserved

SGL Scatter-gather list identifying the buffer containing the fragment.

SwID Identifies the software downloading. For a DDM, a DDM Module Parameter Block, or a DDM Configuration Dialogue Table, the SwID contains two fields from the DDM's module header, as shown below:

31 3 24	23 2 16	15 1 8	7 0 0
OrganizationID		ModuleID	

For downloading any other type, the exact content of this field is implementation specific.

SwType Type of software module downloading, as described in Table 4-6.

SWSize The total number of bytes in the software module.

TotalFragments The total number of fragments in the sequence.

VersionOffset 71h for 32-bit context size and 91h for 64-bit context size.

The available software modules are described in Table 4-6:

Table 4-6 Software Module Types

Value	Description
01h	DDM
02h	DDM Module Parameter Block
03h	DDM Configuration Dialogue Table
11h	IRTOS
12h	IRTOS private module: may be any portion of the IRTOS, depending on the IRTOS implementation.
13h	IRTOS Dialogue Table
22h	IOP private module: Its meaning is implementation specific.
23h	IOP Dialogue Table
others	reserved

4.4.3.28 Software Upload

The host transfers software from the IOP using the ***ExecSwUpload*** message. The software to upload may be one of the following:

- DDM
- DDM Module Parameter Block
- DDM Configuration Dialogue Table
- IRTOS
- IRTOS Private Module
- IRTOS Configuration Dialogue Table
- IOP Private Module
- IOP Configuration Dialogue Table

This message instructs the IOP to place a portion of the requested module in the designated buffer. Only the host can send this message. The normal reply is a default with no payload.

The host can break the module into a sequence of fragments, uploading each using an individual ***ExecSwUpload*** message. All messages in the sequence must have the same transaction context. The IOP uses the transaction context to ensure that all ***ExecSwUpload*** requests are part of the same sequence.

The host must upload only one fragment at a time, wait for the reply, and then request the next fragment of the same module. Only one sequence may be in progress at a time. Interleaving sequences causes an error.

The IOP sends all fragments in order. The first fragment of the sequence (Current Fragment = 1) contains the beginning of the software module. The last fragment (Current Fragment = Total Fragments) contains the last portion of the software module. The IOP aborts any previous uploads if a request arrives specifying a new sequence (Current Fragment = 1). After it sends the first fragment, if the IOP receives any request out of order, it returns an error code in the reply and aborts the operation.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
ExecSwUpload	InitiatorAddress=*001h*	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
UploadFlags	SwType	TotalFragments	CurrentFragment	16 (24)
SWSize				20 (28)
SwID				24 (32)
SGL :				28 (36)

Offset in () signifies offset for 64-bit context fields

Figure 4-40. *ExecSwUpload* Request Message

Fields

CurrentFragment Value indicating the position of the fragment in the sequence. The first fragment is 1, the second fragment is 2, and so forth.

SGL Scatter-gather list identifying the destination buffer of the fragment.

SwID Identifies the software being uploaded. For a DDM, a DDM Module Parameter Block or a DDM Configuration Dialogue Table, the SwID contains two fields from the DDM's module header as shown below:

31 3 24	23 2 16	15 1 8	7 0 0
OrganizationID		ModuleID	

For uploading any other type, the contents of this field are implementation specific.

SwType Type of software module uploading, as described in Table 4-6.

SWSize The number of bytes in the software module. Set to zero if the value is unknown. IOP uses this value to verify correct identification of the module to upload.

TotalFragments The total number of fragments in the sequence.

UploadFlags reserved

VersionOffset 71h for 32-bit context size and 91h for 64-bit context size.

4.4.3.29 Software Remove

The host removes software from the IOP using the ***ExecSwRemove*** message. The message removes one of the following:

- DDM
- DDM Module Parameter Block
- DDM Configuration Dialogue Table
- IRTOS Private Module
- IRTOS Configuration Dialogue Table
- IOP Private Module
- IOP Configuration Dialogue Table

This message instructs the IOP to delete the requested module from its permanent store. The software continues to operate if it is loaded, but does not load the next time the IOP is reset. Only the host can send this message. The normal reply is a default reply with no payload.

The IOP may reject any ***ExecSwRemove*** requests. This message enables a host-based application to clean up the driver store on the IOP.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecSwRemove	InitiatorAddress = 001h		TargetAddress=000h	4
InitiatorContext				8
TransactionContext				12 (16)
RemoveFlags	SwType	reserved		16 (24)
SwSize				20 (28)
SwID				24 (32)

Offset in () signifies offset for 64-bit context fields

Figure 4-41. *ExecSwRemove* Request Message

Fields

SwID — Identifies the software being removed. For a DDM, a DDM Module Parameter Block, or a DDM Configuration Dialogue Table, the SwID contains two fields from the DDM's module header as shown below:

31 3 24	23 2 16	15 1 8	7 0 0
OrganizationID		ModuleID	

For removing any other SwType, the contents of this field is implementation specific.

SwSize — The total number of bytes in the software module. Set to zero if value is unknown. IOP uses this value to verify identification of module to remove.

SwType — Type of software module being removed, as described in Table 4-6.

RemoveFlags reserved

4.4.3.30 System Enable

The ***ExecSysEnable*** message allows the IOP to resume external operations. This command is useful when rebooting the system or when the system reconfigures its I/O bus (to change the address of one or more IOPs, for example). Only the host can send this message. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecSysEnable	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)

Offset in () signifies offset for 64-bit context fields

Figure 4-42. *ExecSysEnable* Request Message

4.4.3.31 System Modify

The ***ExecSysModify*** message resembles the ***ExecIopClear*** message, but it also notifies the IOP that a physical system reconfiguration is imminent. This message causes the IOP to terminate external operations, clearing its input queues and preparing for a system configuration change. This command is useful when rebooting the system or reconfiguring its I/O bus (to change the addresses of IOPs or adapters that the IOP controls, for example). Internal operation of the IOP is affected for all adapters that reside on the system bus. Only the host can send this message. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset	0
ExecSysModify	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)
SGL				16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 4-43. *ExecSysModify* Request Message

Fields

SGL Specifies two single segment buffers: The first is the private memory space declaration and the second is the private I/O space declaration. Any additional buffers in the SGL are ignored. Refer to section 4.4.3.33 ***System*** *Table Set*, for details.

VersionOffset 41h for 32-bit context size and 61h for 64-bit context size.

Before system reconfiguration, the host sends this message with public space declarations so the IOP knows whether to move any of its hidden adapters.

The host is expected to send a ***ExecSysQuiesce*** message to all IOPs before sending a ***ExecSysModify*** message. When it receives this message, the IOP suspends external message service, flushes its inbound message queue, flushes all path information, and reallocates its primary inbound queue. It also suspends operation of any DDM that controls an adapter on the system bus. Once the IOP curtails operation and re-establishes its inbound queue, it replies to the message.

4.4.3.32 System Quiesce

The ***ExecSysQuiesce*** message causes the IOP to make external operations quiescent. This command is useful when rebooting the system or reconfiguring its I/O bus (to change the address of one or more IOPs, for example). Internal operation of the IOP continues normally. Only the host can send this message. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset = 01h	0
ExecSysQuiesce	InitiatorAddress	TargetAddress=000h		4
InitiatorContext				8
TransactionContext				12 (16)

Offset in () signifies offset for 64-bit context fields

Figure 4-44. *ExecSysQuiesce* Request Message

4.4.3.33 System Table Set

Once the host finishes initializing IOPs, it sends an ***ExecSysTabSet*** request to each IOP. This message gives each IOP the identity (location) of the other IOPs in the system, as well as declarations of memory and I/O for private space. This event also takes the IOP from the *HOLD* state to the *READY* state. The normal reply is a default reply with no payload.

31 3 24	23 2 16	15 1 8	7 0 0	offset
MessageSize		MessageFlags	VersionOffset	0
ExecSysTabSet	InitiatorAddress	TargetAddress = 000h		4
InitiatorContext				8
TransactionContext				12 (16)
HostUnitID		reserved	IOP_ID	16 (24)
reserved		reserved	SegmentNumber	20 (28)
SGL				16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 4-45. *ExecSysTabSet* Request Message

Fields

IOP_ID The host assigns a 12-bit number that uniquely identifies the target IOP within this unit. The following values are pre-defined or reserved and are not assigned to an IOP:

000h:	Null IOP_ID	Means the local IOP. Used by DDMs that only function within their local IOP environment.
001h:	Local Host	Reserved for indicating the hosting entity within the unit
FFFh:	Unknown IOP	Denotes all IOPs or IOP_ID unknown.

The host may assign different values than the BIOS.

HostUnitID This is a 16 bit number that uniquely identifies unit hosting the target IOP. The following values are pre-defined or reserved:

0000h	Null	Means the local unit. Used by DDMs and IOPs that only function within their local unit environment. Also used by hosts that are not part of a multi-unit system. This also includes units that initialize prior to resolving its HostUnitID.
FFFFh	Unknown	Denotes all units or HostUnitID unknown.

Typically the BIOS specifies NULL unless it participates in multi-unit protocols. The host may also initially set the value to NULL and later change it via a ***UtilParamsSet*** message.

SegmentNumber This 12-bit number assigned by the local unit uniquely identifies the I_2O Segment within the unit. The following values are pre-defined or reserved:

000h:	Null	Means the local I2O segment. Used by DDMs and IOP's that only function within their local I_2O segment environment. Also used by a host that does not have sufficient information about different I_2O segments.
FFFh:	Unknown	Denotes all segments or SegmentNumber unknown.

This specification allows the BIOS to set the SegmentNumber of each IOP so the host can learn those assignments if it so chooses.

SGL Provides three buffers (as identified by the EndOfBuffer flag). The first contains the I_2O system table as specified in Figure 4-46; the second is the private memory space declaration specified as a single SGL element; and the third is the private I/O space declaration specified as a single SGL element. The IOP ignores any additional buffers in the SGL.

VersionOffset 41h for 32-bit context size and 61h for 64-bit context size.

This message provides three pieces of information to the IOP, as follows:

1. **I_2O System Table (SysTab)**
   Describes the local system (i.e., within the unit) as a set of IOPs and their message attributes. Figure 4-46 provides the format for this table.

2. **Private Memory Space Declaration**
   The private memory space declaration gives the IOP the base address and length of address space where it can configure its hidden adapters. The host may assign the same

private space to all IOPs. The space described by this declaration should constitute a hole in the host's physical memory map.

The private memory space declaration lets the IOP hide adapters from the system and bring adapters on-line after the system is configured. The host learns the requested size by the status returned from the ***ExecStatusGet*** message.

3. **Private I/O Space Declaration**
The private I/O space declaration gives the IOP the base address and length of I/O space where it can configure its hidden adapters. The host may assign the same private space to all IOPs. The space described by this declaration should constitute a hole in the host's physical I/O map.

The private I/O space declaration lets the IOP hide adapters from the system and bring adapters on-line after the system is configured. The host learns the requested size by the status returned from the ***ExecStatusGet*** message.

31 3 24	23 2 16	15 1 8	7 0 0	offset
reserved		SysTabVersion	NumberEntries	0
CurrentChangeIndicator				4
reserved				8
SysTabIopEntry 1 (per **Figure 4-47. System Table Entry**)				16
SysTabIopEntry 2 (per **Figure 4-47. System Table Entry**)				48
SysTabIopEntry 3 (per **Figure 4-47. System Table Entry**)				80
...				
SysTabIopEntry *n* (per **Figure 4-47. System Table Entry**)				

Figure 4-46 System Table Structure

Fields for System Table

CurrentChangeIndicator — Starts at zero and is incremented each time in configuration change is reported

SysTabIopEntry — Describes an IOP. Figure 4-47 describes the structure of this field in more detail. Note, this figure provides for PCI base IOPs. Additional definitions based on other technologies may follow and can be differentiated by the MessengerType field. Each entry is always 32 bytes in length.

NumberEntries — Specifies the number of entries in the table and thus the number of IOPs in the system.

SysTabVersion — Version of I_2O for the system resource manager and, therefore, for this table.

31 3 24	23 2 16	15 1 8	7 0 0	offset
reserved		OrganizationID		0
reserved		reserved	IOP_ID	4
MessengerType	IopState	I2oVersion	SegmentNumber	8
reserved		InboundMessageFrameSize		12
LastChanged				16
IopCapabilities				20
MessengerInfo				24
				28

Figure 4-47. System Table Entry

Fields for System Table Entry

OrganizationID
ID the SIG assigns to the IOP's vendor.

IOP_ID
Arbitrary number assigned by host resource manager to uniquely identify each IOP.

I2oVersion
Version of I_2O specification under which the IOP is operating (from ***ExecStatusGet*** message). This version = 01h.

IopCapabilities
See ***ExecStatusGet*** for values.

IopState
See ExecStatusGet for values. A state other than *OP* means that the IOP is not available. The reception of the ***ExecSysTabSet*** message might cause an IOP state change. This field reflects the expected state after any such change.

InboundMFrameSize
Size of the inbound message frame (in 32-bit words). Minimum allowable size is 16 (i.e., 64 bytes).

LastChanged
Value of CurrentChangeIndicator the last time this messenger reported a configuration change.

MessengerInfo
Content of this field depends on MessengerType. Figure 4-48 specifies the format for MessengerType = Memory Mapped Message Unit.

MessengerType
The only type defined by this version of the specification is:
00h = memory mapped message unit. The IOP should ignore entries with an unknown MessengerType.

31 3 24	23 2 16	15 1 8	7 0 0	offset
InboundMessagePortAddressLow				24
InboundMessagePortAddressHigh				28

Figure 4-48. MessengerInfo for Memory Mapped Message Unit

Fields for Specific System Table Entry

InboundMPortAddressHigh
High 32 bits of system address of the IOP's inbound message FIFO. Not all IOPs support a non zero value.

InboundMPortAddressLow Low 32 bits of system address of the IOP's inbound message FIFO.

4.4.4 Modifying Parameters

The ***UtilParamsGet*** and ***UtilParamsSet*** utility messages specified in Chapter 6 operate on specific Executive class parameters specified in Table 4-7.

Table 4-7. Executive Parameter Groups

GroupNumber	**0000h**
GroupType	SCALAR
Name	***IOP_HARDWARE***
Description	Information to describe the IOP's hardware environment.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 bytes	I2OVendorId	The I$_2$O organization ID assigned to the hardware or system vendor providing the platform.
1	r	2 bytes	ProductID	A value the vendor assigns to identify the product.
2	r	4 bytes	ProcessorMemory	Total amount of RAM memory available for code and data (in bytes).
3	r	4 bytes	PermMemory	Total amount of permanent memory for storing IRTOS and DDMs (in bytes).
4	r	4 bytes	HWCapabilities	Bit-specific field that indicates the capabilities of the IRTOS. A *1* indicates the capability exists: bit 0 Self booting bit 1 IRTOS can be upgraded bit 2 Supports downloading DDMs bit 3 Supports installing DDMs bit 4 Battery-backed RAM
5	r	1 byte	ProcessorType	Type of IOP: 00h Intel 80960 series 01h AMD29000 series 02h Motorola 68000 series 03h ARM series 04h MIPS series 05h Sparc series 06h PowerPC series 07h Alpha series 08h Intel x86 series FFh Other Other values are reserved (for current list see I2O SIG Web site at http://www.i2osig.org/).
6	r	1 byte	ProcessorVersion	Version of processor dependent on ProcessorType: 00h default other values to be determined (for current list visit the I2O SIG Web site at http://www.i2osig.org/).

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0001h**
GroupType	SCALAR
Name	***IOP_MESSAGE_IF***
Description	Provides information regarding the IOP's inbound and outbound message interface.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 bytes	InboundFrameSize	Number of bytes in an inbound message frame.
1	r/w	4 bytes	InboundSizeTarget	Number of bytes of an inbound message frame after next IOP reset.
2	r	4 bytes	InboundMax	Maximum number of inbound message frames that the IOP currently supports.
3	r/w	4 bytes	InboundTarget	Number of inbound message frames that the IOP will support after it resets.
4	r/w	4 bytes	InboundPoolCount	The number of message frames created for the inbound message queue, not counting static frames.
5	r	4 bytes	InboundCurrentFree	The current number of message frames in the inbound Free_List.
6	r	4 bytes	InboundCurrentPost	The current number of message frames in the Inbound Post_List.
7	r	WORD16	StaticCount	Current number of static message frames.
8	r	WORD16	StaticInstanceCount	Current total number of instances (MaxOutstanding) of static message frames that can be in the inbound queue.
9	r/w	WORD16	StaticLimit	Maximum number of static message frames the IOP supports.
10	r/w	WORD16	StaticInstanceLimit	Total number of instances (MaxOutstanding) of static message frames that the IOP supports.
11	r	4 bytes	OutboundFrameSize	Number of bytes in an outbound message frame.
12	r	4 bytes	OutboundMax	Maximum number of outbound message frames that the IOP supports.
13	r/w	4 bytes	OutboundMaxTarget	Number of outbound message frames that the IOP will support after it resets.
14	r	4 bytes	OutboundCurrentFree	The current number of message frames in the outbound Free_List.
15	r	4 bytes	OutboundCurrentPost	The current number of message frames in the Outbound Post_List.
16	r/w	1 byte	InitCode	Indicates who controls the IOP's message interface. This value is modified by the host during different phases of an IOP's initialization sequence. The IOP initially sets this value to zero. This code is set via an ExecOutboundInit message and updated by the host as necessary. The following values are reserved for the identified entity. 00h - 0Fh: No owner,

GroupNumber	**0001h**
GroupType	SCALAR
Name	***IOP_MESSAGE_IF***
Description	Provides information regarding the IOP’s inbound and outbound message interface.

FieldIdx	**(r/w)**	**Field Size**	**Parameter Name**	**Description**
				10h - 1Fh: Reserved for the BIOS vendor 20h - 2Fh: Reserved for the platform OEM BIOS extension 30h - 3Fh: Reserved for ROM BIOS extensions 80h - 8Fh: Reserved for formal Operating System other values: reserved; to be specified later.

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0002h**
GroupType	SCALAR
Name	***EXECUTING_ENVIRONMENT***
Description	Identifies characteristics of the IOP executing environment.

FieldIdx	**(r/w)**	**Field Size**	**Parameter Name**	**Description**
0	r	4 bytes	MemTotal	Total Memory available for execution
1	r	4 bytes	MemFree	Free Memory
2	r	4 bytes	PageSize	Page frame size for the host
3	r	4 bytes	EventQMax	Maximum number of event queues supported
4	r	4 bytes	EventQCurrent	Current number of event queues
5	r	4 bytes	DdmLoadMax	Maximum number of drivers that can be loaded

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0003h**
GroupType	TABLE
Name	***EXECUTING_DDM_LIST***
Description	Identifies drivers that are loaded in the IOP’s executing environment.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 bytes	DdmTID	The TID of the executing DDM (the most significant four bits are zeros).
1	r	1 byte	ModuleType	Module type extracted from Table 4-6. 00h Other 01h Downloaded DDM 22h Embedded DDM
2	r	1 byte	reserved1	reserved
3	r	2 bytes	I2oVendorID	I$_2$O Organization ID for the DDM vendor
4	r	2 bytes	ModuleID	Module ID assigned by the DDM vendor
5	r	24 Bytes	ModuleName	Module name (24 ASCII characters) from header.
6	r	4 bytes	ModuleVersion	Module version (four ASCII characters) from header.
7	r	4 bytes	DataSize	Current memory use, in bytes
8	r	4 bytes	CodeSize	Code size, in bytes

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0004h**
GroupType	SCALAR
Name	***DRIVER_STORE***
Description	Identifies characteristics of the IOP’s permanent storage media.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 bytes	ModuleLimit	Maximum Number of driver modules that can be stored
1	r	4 bytes	ModuleCount	Current number of driver modules saved in IOP’s store
2	r	4 bytes	CurrentSpace	Number of bytes consumed by driver modules saved in IOP’s store
3	r	4 bytes	FreeSpace	Number of bytes left in the IOP’s permanent store

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0005h**
GroupType	TABLE
Name	***DRIVER_STORE_TABLE***
Description	Identifies the drivers stored in the IOP permanent storage media.

FieldIdx	**(r/w)**	**Field Size**	**Parameter Name**	**Description**
0	r	2 bytes	StoredDdmIndex	Unique identifier for each stored DDM
1	r	1 byte	ModuleType	Module type extracted from Table 4-6 00h Other 01h Downloaded DDM 22h Embedded DDM
	r	1 byte	reserve1	reserved
0	r	2 bytes	I2oVendorID	I$_2$O Organization ID for the vendor from header
1	r	2 bytes	ModuleID	Vendor-assigned module ID from header
2	r	24 bytes	ModuleName	Module name (24 ASCII characters) from header
3	r	4 bytes	ModuleVersion	Module version (four ASCII characters) from header
4	r	2 bytes	DateDay	Day of module date (two ASCII characters) from header
5	r	2 bytes	DateMonth	Month of module date (two ASCII characters) from header
6	r	4 bytes	DateYear	Year of module date (four ASCII characters) from header
7	r	4 bytes	ModuleSize	Size of stored image, in bytes
8	r	4 bytes	MpbSize	Size of stored module parameter block, in bytes
9	r	4 bytes	ModuleFlags	reserved

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0006h**
GroupType	TABLE
Name	***IOP_BUS_ATTRIBUTES***
Description	Specifies the number and type of I/O busses supported by the IOP.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 bytes	BusID	Bus ID assigned by the IOP
1	r	1 byte	BusType	Bus type
2	r	1 byte	MaxAdapters	Maximum number of adapters
3	r	1 byte	AdapterCount	Current number of adapters
4	r	1 byte	BusAttributes	Bus Attributes (bridged, private, system) 00h = System bus - host can access this bus; adapters on this bus can access system memory. 01h = Bridged to system bus - host cannot access this bus but adapters on this bus can access system memory. 02h = Private, no system access - host cannot access this bus; adapters on this bus cannot access system memory.

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0007h**
GroupType	SCALAR
Name	***IOP_SW_ATTRIBUTES***
Description	Provides information about the IOP's operating system.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 bytes	I2oVendorId	The I_2O organization ID assigned to the vendor providing the IRTOS
1	r	2 bytes	ProductID	A value the vendor assigns to identify the product
2	r	4 bytes	CodeSize	Code Size (in bytes) of the IOP operating system
3	r	4 bytes	SWCapabilities	A set of flags indicating the capabilities of the IRTOS: bit 0 IRTOS is I_2O compliant bit 1 IRTOS can be upgraded bit 2 Supports downloading DDMs bit 3 Supports installing DDMs

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0100h**
GroupType	TABLE
Name	***HARDWARE_RESOURCE_TABLE***
Description	Hardware Resource Table describes adapters that the IOP controls or can control

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 bytes	AdapterID	Arbitrary value assigned to the adapter by the IOP used to identify it
1	r	2 bytes	StateInfo	AdapterState plus Local TID of the HDM to which this adapter is assigned. ControllingTID is the least significant 12 bits, and AdapterState is the most significant four bits.
2	r	1 byte	BusNumber	Arbitrary value the IOP assigns that identifies the physical bus where the adapter resides
3	r	1 byte	BusType	Indicates the type of expansion bus. For values, see *Common Structures for Adapters* in Chapter 3.
4	r	8 bytes	PhysicalLocation	Eight bytes of data that identify the physical device or function. Format depends on BusType, and those variations are defined in *Common Structures for Adapters*, in Chapter 3.
5	r	4 bytes	MemorySpace	Amount of memory space consumed by the adapter
6	r	4 bytes	IoSpace	Amount of I/O space consumed by the adapter

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0101h**
GroupType	SCALAR
Name	***LCT_SCALAR***
Description	Scalar (non-LCT entry) values associated with the LCT

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 bytes	BootDevice	TID of the device that booted the OS. Set to zero if none or unknown. This value is set by the BIOS via the ***ExecBootDeviceSet*** message.
1	r	4 bytes	IOPFlags	Bit 0: Set indicates that the IOP requests a configuration dialogue All other bits reserved
2	r	4 bytes	CurrentChangeIndicator	Initialized to 0 and incremented before the table is returned, only if the table changed since the last table read response.

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0102h**
GroupType	TABLE
Name	***LCT_TABLE***
Description	Logical Configuration Table

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 bytes	LocalTID	Local Target ID assigned by IOP to this device (the most significant four bits are zeros)
1	r	2 bytes	UserTID	TID (the most significant four bits are zeros) of the primary service user of this device. Established by connection setup (UtilClaim), it indicates the OSM or ISM to which this resource is dedicated. The value of 0FFFh indicates that the resource is not allocated. A value other than 0FFFh indicates that the device is reserved.
2	r	2 bytes	ParentTID	TID (the most significant four bits are zeros) of the DDM or device that created, registered and manages this I/O device.
3	r	2 bytes	DdmTID	TID of the DDM under which this device was created
4	r	4 bytes	ChangeIndicator	Value of CurrentChangeIndicator last time this entry was updated
5	r	4 bytes	DeviceFlags	Bit-specific field that identifies the device's characteristics and capabilities. Bit 0: Set to indicate that the device requests a configuration dialogue. Bit 1: Set if the device can concurrently support more than one user All other bits are reserved
6	r	4 bytes	ClassID	Message class of this device. Messages sent to the LocalTID must confirm to this class definition. See Chapter 6 for the definition of ClassID.
7	r	4 bytes	SubClass	Defined by the message class
8	r	8 bytes	IdentityTag	Part of the serial number that uniquely identifies a device. This field is always eight bytes long and does not include the SNLen or SNFormat fields. If the serial number is less than 64 bits, it is pre-padded with zeros. If the serial number exceeds eight bytes, it is truncated to the lowest order eight bytes (i.e., the bits that provide unique identity). This field is used to match system configuration with the device. If no serial number is known, this field contains all zeros.
9	r	4 bytes	EventCapabilities	Each bit of this field corresponds to the same bit in the EventMask field of the UtilEventRegister message. A 1 indicates that the DDM can generate that type of event.
10	r	1 byte	BiosInfo	Identifier used to correlate a device with any connections the BIOS creates. For example, when a BIOS extension hooks INT13 for a storage device and is assigned drive ID 81h, this field is set to 81h to notify the OS not to call the BIOS for drive ID 81h. The default value 0FFh indicates that the device is not the subject of a BIOS function call. This value is set by the BIOS via the ***ExecBiosInfoSet*** message.

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0103h**			
GroupType	TABLE			
Name	***SYSTEM_TABLE***			
Description	Describes the local I_2O system as a set of IOPs and their Attributes.			

FieldIdx	**(r/w)**	**Field Size**	**Parameter Name**	**Description**
0	r	2 bytes	IOP_ID	Arbitrary number the host resource manager assigns to uniquely identify each IOP within the unit (from ***ExecSysTabSet*** message).
1	r	2 bytes	OrganizationID	ID the SIG assigns to the vendor of the IOP.
2	r/w	2 bytes	SegmentNumber	Segment ID (from ***ExecSysTabSet*** message).
3	r	1 byte	I2OVersion	Version of the I_2O specification under which the IOP operates (the most significant four bits are zeros).
4	r	1 byte	IopState	See ExecStatusGet message for values
5	r	1 byte	MessengerType	The only type defined by this version of the specification is: 00h Memory-mapped message unit
6	r	1 byte	reserved	
7	r	4 bytes	InboundMessagePortAddress	System address of the IOP's inbound message FIFO.
8	r	2 bytes	InboundMessageFrameSize	Size of the inbound message frame (in 32-bit words). Minimum size is 16 (i.e., 64 bytes).
9	r	4 bytes	IopCapabilities	See ***ExecStatusGet*** message.
10	r	8 bytes	MessengerInfo	See ***ExecSysTabSet*** message

Table 4-7. Executive Parameter Groups (continued)

GroupNumber	**0104h**
GroupType	TABLE
Name	***EXTERNAL_CONNECTION_TABLE***
Description	External Connection Table

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 bytes	LocalAliasTID	TID (the most significant four bits are zeros) assigned by this IOP, uniquely identifying the device to DDMs on this IOP.
1	r	2 bytes	RemoteTID	TID (the most significant four bits are zeros) identifying the device at the far end of the connection. This is the TID from the remote IOP's logical configuration table.
2	r	2 bytes	RemoteIOP	IOP_ID identifying the other IOP. Assigned by the host from the system table. Upper four bits are reserved.
3	r	2 bytes	RemoteUnitID	HostUnitID of the other IOP.
4	r	1 byte	Flags	Bit 0: Initiator flag: 0 Remote IOP created connection 1 This IOP created connection Bits 1-7 are reserved.
5	r	1 byte	reserved1	

4.5 I_2O Behavioral Model

Unless otherwise specified, the target of a request responds with one or more replies. Some functions allow a single reply to acknowledge multiple requests. A reply is returned to the originator as identified by the InitiatorAddress field of the request. The TargetAddress field of the reply contains the TID of the device sending the reply, which was specified as the TargetAddress in the original request. If the host issues requests, the InitiatorAddress and TargetAddress fields of the request are copied to the respective fields of the reply.

The InitiatorContext is always copied unchanged from the request to the reply. The payload of the request depends on the value of the Function field and the class registered to the TargetAddress. The payload of the reply depends on the request Function field, the class registered to the TargetAddress, and the status of the reply. It is the requester's responsibility to correlate a reply to the appropriate request, using the TransactionContext field.

4.5.1 I_2O System Initialization

To bring the I_2O system to a fully *operational state*, the host must provide the basic initialization. This is typically a two-phase process. The first phase involves the BIOS, and the second phase involves the OS. The BIOS is usually interested in an IOP only for booting the operating system. It is the operating system that brings the IOPs to full operation and enables peer-to-peer operation. During the OS boot, the OS may reinitialize the I_2O sub-system to establish its own system configuration.

This section describes the protocol for initializing the I_2O system. It describes bringing IOPs online from a power-on state, the interaction with the BIOS, and rebooting the system.

4.5.1.1 System Initialization

Figure 4-49 illustrates an initialization sequence typical of both the BIOS and OS. I_2O subsystem initialization makes the following assumptions:

- The IRTOS and the DDMs are all stored in the IOP permanent store.
- All I_2O devices have been previously configured.
- No problems arise during initialization.

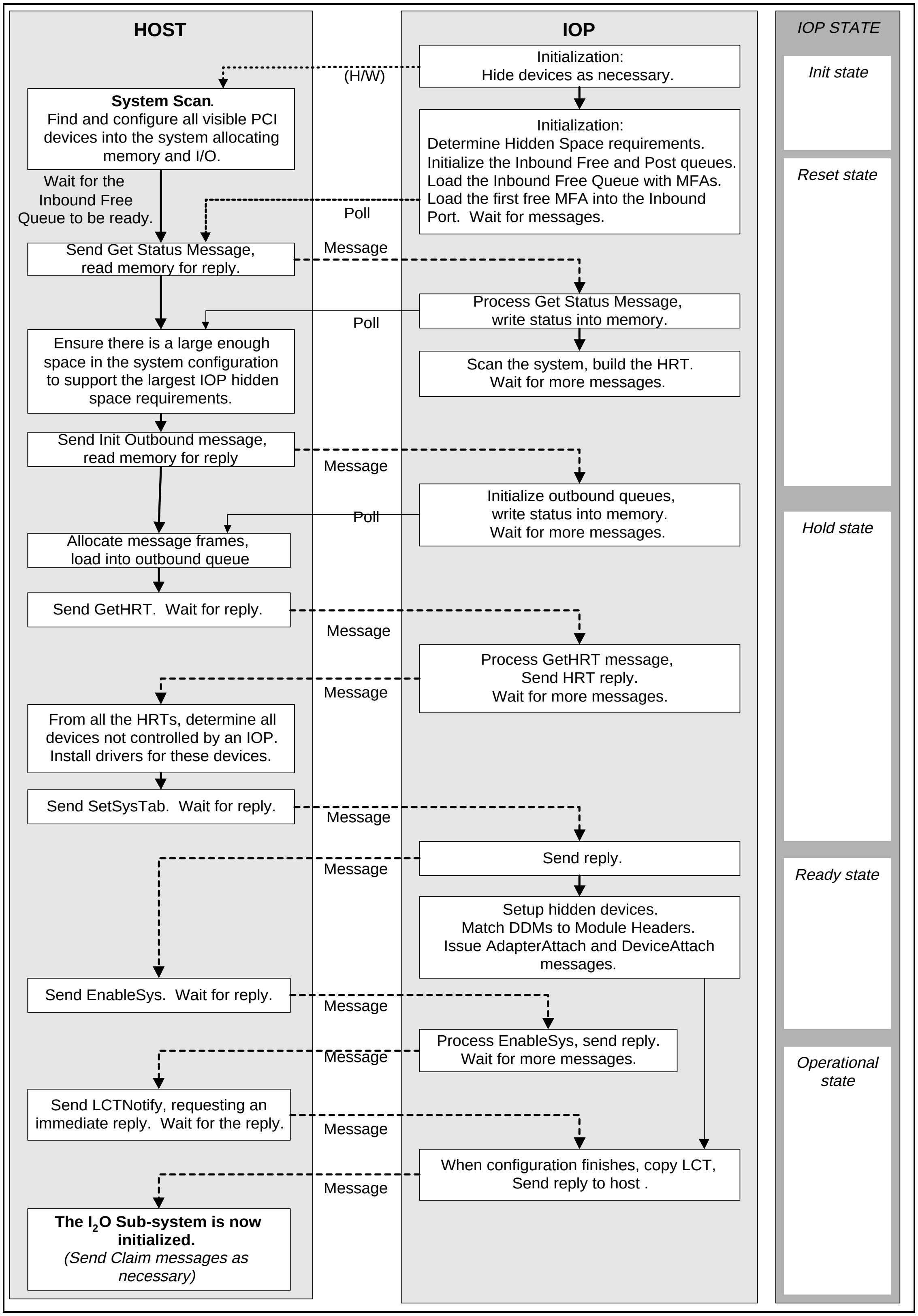

HOST
IOP
IOP STATE
Initialization:
Hide devices as necessary.
(H/W)
Init state
System Scan.
Find and configure all visible PCI devices into the system allocating memory and I/O.
Initialization:
Determine Hidden Space requirements.
Initialize the Inbound Free and Post queues.
Load the Inbound Free Queue with MFAs.
Load the first free MFA into the Inbound Port. Wait for messages.
Reset state
Wait for the Inbound Free Queue to be ready.
Poll
Send Get Status Message, read memory for reply.
Message
Process Get Status Message, write status into memory.
Poll
Ensure there is a large enough space in the system configuration to support the largest IOP hidden space requirements.
Scan the system, build the HRT.
Wait for more messages.
Send Init Outbound message, read memory for reply
Message
Initialize outbound queues, write status into memory.
Wait for more messages.
Poll
Allocate message frames, load into outbound queue
Hold state
Send GetHRT. Wait for reply.
Message
Process GetHRT message, Send HRT reply.
Wait for more messages.
Message
From all the HRTs, determine all devices not controlled by an IOP. Install drivers for these devices.
Send SetSysTab. Wait for reply.
Message
Send reply.
Message
Ready state
Setup hidden devices.
Match DDMs to Module Headers.
Issue AdapterAttach and DeviceAttach messages.
Send EnableSys. Wait for reply.
Message
Process EnableSys, send reply.
Wait for more messages.
Message
Operational state
Send LCTNotify, requesting an immediate reply. Wait for the reply.
Message
When configuration finishes, copy LCT, Send reply to host .
Message
The I2O Sub-system is now initialized.
(Send Claim messages as necessary)

Figure 4-49. Typical System Initialization

The key to initializing an I$_2$O sub-system is understanding the points where the host and the IOP must synchronize. These points are described below:

1. As shown in the Figure 4-49, both the host and the IOP begin initialization in parallel:
    - The IOP must hide any hidden adapters before the host performs its bus scan (the host must never see hidden adapters). The configuration establishes the adapters the IOP hides. The IOP preserves that list from boot to boot.
    - Once the IOP hides adapters, it may scan them to determine their system memory and I/O requirements. The IOP must not scan any adapters on system buses, since it would interfere with the host system scan.
    - The IOP initializes its Inbound Free and Post queues, allocates a number of inbound message frames, and loads the Inbound Free Queue with the MFAs for those frames. As a result, reading the Inbound Port reads the first MFA from the Inbound Free Queue.
2. The host scans the system I/O bus, configuring all visible adapters. It then finds each IOP's inbound and outbound FIFOs, using the method discussed in section 4.2.1.4. Before the IOP initializes, the host reads only FFFF-FFFFh from the IOP's inbound free list FIFO. Once IOP loads its inbound queue, the host reads a valid MFA. The IOP is in the *Reset* state and can receive messages from the host.
3. The ***ExecStatusGet*** request is typically the first message the host sends to each IOP. Since the IOP's outbound queue might not yet be initialized, the IOP does not reply. Instead, the host polls the status buffer specified in the request to determine when the IOP responds. From this synchronization point, the host determines whether the IOP is functional and responding to messages.

    Once the IOP receives the first ***ExecStatusGet*** message, it knows the host has completed its system scan. It is now safe for the IOP to perform its own system scan and build its Hardware Resource Table.

    The ***ExecStatusGet*** request identifies the hidden memory and I/O space needs of the IOP. The host needs to find a hole in its memory and I/O map to accommodate the largest request. If there is not a space large enough to meet the hidden adapters' needs, the host may need to reconfigure the system. If the host cannot provide as large a space as required, it should set up one as large as possible. The final system configuration is saved in the System Table structure, which is sent to each IOP after all IOPs are configured.
4. The host sends an ***ExecOutboundInit*** request to each IOP. This request causes the IOP to flush its outbound FIFOs, which might be loaded from the BIOS session or from a previous OS session. Since the outbound queue might not yet be initialized, the IOP does not reply. Instead, the host determines when the operation completes by polling the status buffer specified in the request.

    Receiving this message causes the IOP to transition to the *Hold* state. When it detects the response, the host creates a number of empty message frames and primes the IOP's outbound queue by writing the MFA of each frame to the IOP's outbound FIFO. The IOP can now reply to the host.
5. The host sends an ***ExecHrtGet*** request to each IOP. When it receives the request, the IOP returns its Hardware Resource Table (HRT). This table tells the host which adapters are assigned to the IOP and which unassigned adapters the IOP can control. If the host detects

an adapter assigned to multiple IOPs, the host must resolve the conflict. The host must also prevent its drivers from initializing and controlling adapters assigned to an IOP.

- The host can facilitate changes to an IOP's HRT by using the ***ExecAdapterAssign*** and ***ExecAdapterRelease*** messages.
- IOPs must not initialize assigned adapters until after the host can resolve such conflicts (as described below).
- After receiving the Hardware Resource Table from each IOP in the system, the host may now install device drivers for all adapters not controlled by an IOP.

6. As the host discovers and initializes IOPs, it builds a list of IOPs and their FIFO addresses. When the list, called the *I_2O system table*, is complete, the host sends an ***ExecSysTabSet*** request to each IOP providing the System Table and assigning an IOP_ID. Certain messages uniquely identify an IOP using its IOP_ID.

   Receiving the ***ExecSysTabSet*** message:

   - causes the IOP to transition to the *Ready* state.
   - gives the IOP the declaration for private space, allowing the IOP to configure hidden adapters.
   - signals the IOP that any adapter conflicts are resolved and that the IOP may now initialize its adapters.

   The IOP initializes its adapters by matching them against module headers and issuing ***DdmAdapterAttach*** messages to matching DDMs. As a result of the ***DdmAdapterAttach***, the DDMs registers devices. As devices are registered, the IOP matches the new device against module headers and issues ***DdmDeviceAttach*** messages to matching DDMs. This last step repeats until all ***DdmAdapterAttach*** and ***DdmDeviceAttach*** requests conclude and no new devices are registered.

7. Once the host sends the system table to all IOPs, it sends an ***ExecSysEnable*** request to each IOP. This causes the IOP to transition to the *Operational* State. Receiving the ***ExecSysEnable*** message permits the IOP to establish peer-to-peer communication with any IOP listed as operational in the ***ExecSysTabSet*** message.

8. The host completes system initialization by gathering Logical Configuration Tables from all IOPs, via the ***ExecLctNotify*** request message. The host learns the size of the IOP's LCT from its response to the ***ExecStatusGet*** message (the IOP must fill in the Expected Size of LCT field with an estimated final LCT size). If, before sending the ***ExecLctNotify*** message, the host wants the exact size of the LCT, it may send another ***ExecStatusGet*** request.

   The Logical Configuration Table lists all the IOP's registered devices and their availability. The IOP does not reply until its initialization completes (i.e., all DDMs conclude their initialization with no additional devices created). From the reply, the host determines the I_2O class for each device and which devices are unclaimed.

The I_2O sub-system is now fully initialized and can accept class-specific messages. The host may send ***UtilClaim*** requests to I_2O devices that are listed as unclaimed in the Logical Configuration Table.

4.5.1.2 System Re-Initialization

During a warm-boot, reset or OS initialization, the host may wish to reset the IOP to ensure it is in a known state. This can be accomplished with a ***ExecIopReset*** request. The host and the IOP may then proceed with the typical system initialization described above.

4.5.1.3 Abnormal System Initialization

An abnormal system initialization is any that varies from the sequence outlined in section 4.5.1.1. There are many potential causes for an abnormal initialization.

This specification does not explain how an IOP or an I_2O sub-system recovers from an abnormal initialization. However, a number of messages aid in system recovery, as does the Configuration Dialogue facility.

4.5.2 BIOS Considerations

Typically, the system BIOS is concerned only with storage class devices and remote load devices, such as a network adapter that provides remote initial program load (RIPL). Other classes of devices and peer-to-peer operation are not necessary to bring up the system. Newer operating systems often provide their own transport and OSMs for I_2O, rather than relying on BIOS facilities. However, the BIOS must initialize an IOP to boot the operating system when the IOP controls the boot device.

4.5.2.1 Bootstrap Process

Booting from an IOP requires either that the main BIOS be I_2O aware, or that the IOP provide a BIOS extension. Control of an IOP by the main BIOS and a BIOS extension cannot coexist. Since BIOS extensions execute after the main BIOS, the BIOS extension must test whether the system’s BIOS initialized the IOP and, if so, yield to the system’s BIOS. Refer to section 3.1.1 for more details.

Both I_2O-aware system BIOS and BIOS extensions for I_2O adapters must provide the following features:

- The ability to send and receive I_2O messages using the queuing model described in section 4.2.2.
- The ability to initialize the IOP as described in section 4.5.1.
- Modified Int13H calls that use I_2O messaging to access storage class devices.
- User notification that the IOP requests a configuration dialogue session.

An I_2O-aware system BIOS must also be able to only execute BIOS extensions for adapters not controlled by an IOP. The BIOS makes this decision by examining the Hardware Resource Tables from all IOPs in the system.

A BIOS extension for an IOP must also have the following features:

- The ability to determine if the system’s BIOS has initialized the IOP. If so, the BIOS extension must not interfere.
- Initializes only IOPs for which it is specifically developed.

This specification does not address BIOS implementation.

The BIOS uses the IOP's outbound message FIFO, but it may not have that facility available after the OS boots, since the OS claims it exclusively. However, the OS depends on the BIOS to provide access to the boot device until the OS is operational. To smooth the transition from BIOS to OS, the BIOS must inform the OS about I_2O devices accessible through the BIOS. To be more specific, a field in the IOP's logical configuration table is reserved for the BIOS to identify devices accessed by a BIOS function call. The BIOS sends an ***ExecBiosInfoSet*** request to an IOP to set the BiosInfo field in the IOP's logical configuration table. The value identifies the logical unit number assigned by the BIOS (i.e., the value specified in the BIOS function call that identifies the device). For example, the BiosInfo field in the entry for drive C: would contain 80H and the entry for drive D: would contain 81H.

If the BIOS selects an I_2O device as the boot device, the BIOS modifies the BootDevice field of the logical configuration table to indicate the TID of the boot device. This occurs when the BIOS sends an ***ExecBootDeviceSet*** request to the IOP.

Since an OS may decide to reset the IOP during the OS boot process, the IOP must preserve the BootDevice and BiosInfo fields of the LCT.

4.5.2.2 Remote Boot

Remote booting requires the HDM for a LAN adapter, or any other device that can boot remotely (i.e., RIPL), to register both under its native class and as a remote boot class device. The RIPL HDM sends the appropriate boot-me packets across the LAN to the boot server, and the server returns an executable image to the RIPL HDM. The HDM, in turn, provides the executable image when responding to the system BIOS boot requests.

The booted operating system can detect that the boot device was a network adapter by inspecting the IOP's logical configuration table (see the discussion of ***ExecBootDeviceSet*** in 4.5.2.1). The OS queries the HDM for the particular remote boot parameters, such as the boot server address.

Thus, a single HDM with two registered devices, RIPL and network, controls the hardware. Switching from BIOS boot driver control to network driver control is easily resolved within the HDM.

4.5.3 Runtime Considerations

While the system is running, the following host and IOP behaviors may occur:

1. Any time the host changes the system table, it sends another ***ExecSysTabSet*** request to each IOP. From this information, the IOP learns about all other IOPs in the system. Receiving the ***ExecSysTabSet*** request indicates that, when the IOP is in the *Operational* state, it can establish peer-to-peer communication with any IOP listed as being in the *Operational* state.
2. The host may send an ***ExecLctNotify*** request to each IOP. The IOP replies to this message when its logical configuration changes. The IOP uses this reply to ask the host to initiate a Configuration Dialogue session.
3. The host registers for each event it wants to monitor by sending the ***UtilEventRegister*** request to any TID, including the IOP. The target sends an ***UtilEventRegister*** reply to the host whenever the event occurs.

4. As an IOP connects with devices on other IOPs, it builds an external connection table. The host can get the IOP's external connection table by sending an ***UtilParamsGet*** request. See the discussion on system recovery in section 4.5.6 for more information on using this table.

4.5.4 Establishing Paths and Connections

A path is characterized by the address of the inbound message queue (inbound port) of the IOP where the target resides. All messages to a particular IOP use the same path, so the reply path must be explicitly identified. A connection is characterized by the ID of the module at each end of the connection, plus the paths used to deliver messages in both directions. When a connection is established, the IOP creates a TID that it uses to identify both the path and the device registered on a remote IOP. This TID is referred to as an alias, because it can differ from the TID assigned by the IOP hosting the device. By exchanging aliases, IOPs can uniquely identify the source and target devices when exchanging messages.

By design, paths from host OSMs to all I$_2$O registered devices exist by default and do not require aliases. That means that an OSM can send requests to any TID without formally establishing a path or connection. A peer-to-peer connection between IOPs, on the other hand, must formally establish a connection before sending messages.

Before a connection is established between devices, a path between the IOP's executives must be established. The ***ExecIopConnect*** message accomplishes that task by exchanging aliases to use for sending executive messages between the two IOPs. This is required before any other executive messages are exchanged. The system table provides all information necessary to compose a ***ExecIopConnect*** message.

Once the executive path is established, either IOP can request the other IOP's logical configuration table, using the ***ExecLctNotify*** message. This table provides all the information needed to identify devices registered on the remote IOP and thus, to generate a ***ExecConnSetup*** message. The ***ExecConnSetup*** can establish either a true *PEER-TO-PEER* connection by which either end can send requests. Or, it can set up a *CLIENT/SERVER* connection that supports requests in only one direction, that is the same direction as the ***ExecConnSetup*** request. The result of the ***ExecConnSetup*** transaction provides each IOP with the aliases necessary to exchange messages between the two devices.

4.5.5 Configuring the IOP

The following describes the messages that configure the IOP and its modules.

4.5.5.1 Managing the IOP

The ***UtilParamsGet*** and ***UtilParamsSet*** messages let the host manage the operating parameters of the IOP. These operating parameters are general to all IOPs. Static operating parameters, which can be modified only before a session starts, are read-only and must be changed by the configuration utility.

The configuration utility is invoked when the host sends a ***UtilConfigDialog*** request. The reply to a configuration dialogue request is a set of instructions for displaying configuration information on the console, prompting the user for input, accessing a floppy disk drive, and terminating the session. This dialogue modifies the IOP's profile, establishing user-configurable parameters, such as the number of inbound message frames.

The host initiates the configuration dialogue at any time. The IOP indicates the need for a configuration dialogue by setting the appropriate flag bit in the logical configuration table. The configuration dialogue also applies to each module loaded on the IOP, but the dialogue is invoked independently for each device, using a ***UtilConfigDialog*** request addressed to it. Again, a flag bit for each device exists in the logical configuration table to indicate that a configuration dialogue is requested. Setting the flag causes a response to the ***ExecLctNotify*** request, if one was posted. Resetting the flag does not.

4.5.5.2 Installing, Loading, and Configuring Modules

Several messages support installing and loading modules.

Installation primarily stores the module's executable code in the IOP's permanent store so that it can load next time the IOP initializes. The IOP determines whether an installed module needs to be loaded. Loading includes placing the module's executable code in IOP main memory and invoking its initialization program. When a DDM is installed, the IOP creates a module parameter block, if one is not supplied. The module uses this file to store locally-configured parameters.

The ***ExecSwDownload*** message provides the module's executable code and, optionally, the DDM's module parameter block. When the module parameter block is not included, the IOP creates a null parameter block. In either case, when the module is loaded, the IOP calls the modules initialization routine with a pointer to the module parameter block. The module can set its configuration dialogue request flag. The result of the configuration dialogue defines a proper module parameter block that stores user-selected options.

When a module is first installed, it is tagged as *experimental*. The first time the module is loaded, the IOP changes the *experimental* tag to *suspect*. If the host sends an ***ExecConfigValidate*** message, the IOP changes the tag to *valid*. The IOP may set its own configuration flag and prompt the user to accept, reject or defer the experimental module. If the user does not accept the module before the IOP is booted again, then the experimental module is marked *REJECTED* and not loaded. The IOP must then provide a configuration dialogue that warns the user of the rejection. The IOP may delete rejected modules from its permanent store at any time.

When a replacement module is installed, the existing module is tagged *old* and is not loaded. If the experimental module is *ACCEPTED*, then the old module is removed. Otherwise, the IOP loads and tags the old module as valid the next time the IOP boots, while tagging the suspect module as rejected.

The ***ExecDdmDestroy*** and ***ExecSwRemove*** messages cause modules to be unloaded and un-installed, respectively.

The ***ExecAdapterAssign*** message assigns an adapter residing on the system bus to be controlled by a HDM on the IOP.

The ***ExecDeviceAssign*** message is synonymous to the ***ExecAdapterAssign*** message, except that it ties a registered device to an ISM. This message can link an ISM on one IOP with a device registered on another, as well as establish a peer-to-peer link between modules on different IOPs.

The ***ExecAdapterRelease*** and ***ExecDeviceRelease*** messages invoke the IOP to release the assignments made by the ***ExecAdapterAssign*** and ***ExecDeviceAssign*** transactions.

For each adapter assigned to a specific DDM, the IOP issues a ***DdmAdapterAttach*** to the specific DDM. The DDM either accepts or rejects the assignment. For adapters that are rejected or not assigned to a specific DDM, the IOP searches the module headers of DDMs looking for one that lists that adapter. When it finds the DDM, the IOP loads and initializes it, if necessary, and issues a ***DdmAdapterAttach*** to that DDM.

During the ***DdmAdapterAttach***, the DDM should initialize the adapter, register an adapter class device for each I/O port, and register a peripheral class device for each physical device found. The DDM does not reply to the ***DdmAdapterAttach*** until it registers its devices for that adapter.

As devices are registered, the IOP checks whether the device is assigned to a specific DDM. If so, the IOP issues a ***DdmDeviceAttach*** to the specific DDM. The DDM either accepts or rejects the assignment.

During the ***DdmDeviceAttach***, the DDM should claim the device and register additional devices as appropriate. The DDM should not reply to the ***DdmDeviceAttach*** until it claims and registers any new devices.

After the assigned DDM replies to the ***DdmDeviceAttach***, the IOP searches the module headers of DDMs looking for additional DDMs that list that device.As if finds a match, the IOP issues a ***DdmDeviceAttach*** to those DDMs. This operation is different from ***DdmAdapterAttach*** because the adapter can have only one controlling entity while devices may serve multiple users.

Once all DDMs reply to the ***DdmAdapterAttach*** and ***DdmDeviceAttach*** messages, the IOP concludes building its LCT and replies to ***ExecLctNotify*** requests.

DDMs such as RAID, that combine several devices into a single abstracted device, should register that device as soon as it determines that the abstracted device will exist. In the case of RAID, its volume still exists even if one or more drives are missing. The DDM may issue an ***ExecLctNotify*** request to the IOP when the DDM is first initialized, knowing that the IOP replies only after all configuration is complete. If the DDM expects additional ***DdmDeviceAttach*** messages for a particular device, the DDM may defer processing base class messages to that device until the ***ExecLctNotify*** reply. The ***ExecLctNotify*** reply indicates that the DDM must perform appropriate recovery or reconfiguration.

For example, if a RAID DDM combines 5 drives into a RAID volume, but only requires 4 of those drives for operation. When DDM receives the ***DdmDeviceAttach*** for the 4th drive it registers the RAID volume but defers base class messages until the 5th drive is attached. If the DDM receives the ***ExecLctNotify*** reply before the ***DdmDeviceAttach*** for the 5th drive, then it rebuilds the volume as necessary.

4.5.6 System Recovery

The host has several recovery messages. The ***UtilEventRegister*** message allows an IOP to notify the host when another IOP misbehaves or does not respond. The ***UtilParamsGet*** message for the XCT parameter group tells the host how the IOPs are interconnected. The host controls the entire system point by point.

When the host reconfigures the system (changes the addresses of the IOPs, for example), it sends the ***ExecSysModify*** command to all IOPs before making any changes and issues a ***ExecSysTabSet*** to each IOP, followed by an ***ExecSysEnable*** message. If the host does not change the system address of the IOP or any of its adapters, it may use the ***ExecIopClear*** message in lieu of the ***ExecSysModify*** message.

When the host restarts a single IOP, it sends a ***ExecSysQuiesce*** to the faulty IOP and a ***ExecPathQuiesce*** message to all other IOPs. It issues either an ***ExecIopClear*** or ***ExecIopReset*** to the faulty IOP. Once the IOP is operating, the host sends the ***ExecSysTabSet*** message to all IOPs to re-enable their connections and rebuilds external connections, as necessary, with the ***ExecDeviceAssign*** message.

Individual connections can be managed with the ***ExecDdmQuiesce*** and ***ExecDdmSuspend*** messages.

5 I_2O Core Specification

The previous chapter described the IOP from the system viewpoint. This chapter presents the IOP from the perspective of the DDM: the core. The core is the environment that the IOP creates for loadable DDMs. This core specification describes the following:

- the operation of the IOP from the viewpoint of its client (loadable) DDMs
- the operating environment for those client DDMs
- the requirements for a driver to be a loadable DDM.

5.1 Conceptual Overview

The primary functions of the IOP core provide an operating environment for its DDMs and communication between those and other modules, both local and remote. Remote refers to OSMs resident on the host or DDMs resident on another IOP.

The I_2O core specification defines the following:

- **Installation** saves the DDM in the IOP's local permanent store, identifies adapters and devices that the DDM can control, builds a module parameter block, and stores that block in the IOP's local store.
- **DDM configuration** assigns adapters to HDMs, assigns devices to ISMs, and updates the IOP's configuration tables.
- **Initialization** loads the DDM, initializes it, attaches adapters to it, registers the devices the DDM creates, attaches those devices to other DDMs as appropriate, and updates the IOP's logical configuration table.
- **Message service** enables the DDM to query the IOP's configuration database, create connections, and deliver messages on those connections.
- **Flow control** meters the flow of messages to DDMs. If a slow device clogs the queue (i.e., its event queue backs up, starving the IOP's free list), the IOP must reject messages to that device. The IOP does this by replying immediately to the host with a failed status and reclaiming the original message frame. This version of the document makes no requirement for flow control.
- **Transport service** gives the DDM system access and access to adapters (i.e., bus access).
- The **execution environment** provides the real-time operating system functions, such as thread management and memory allocation.

Registering an I_2O device places the information about that device in the IOP's logical configuration table. When the DDM creates an I/O object, the IRTOS registers it as an I_2O device.

Starting a DDM includes installing, loading, and initializing. Installation places a driver in an IOP's permanent storage, so the IOP can load and initialize the driver without host intervention. DDMs can also be loaded from the host without being installed. Loading places the driver code in executable memory and initializing executes that code.

NOTE

The terms DDM, module, and driver are nearly synonymous. This specification generally uses DDM. The choice of one term over the other does not convey any particular meaning, except that module and driver may also refer to the OSM portion as well.

5.1.1 Installing DDMs

Installation is a one-time, initial download of the DDM to the IOP. DDMs are installed and uninstalled as a result of messages from the host (see Chapter 4, *I_2O Shell Interface Specification*). The IOP stores the DDM in its permanent storage. The components of a DDM are the executable code, module descriptor tables, and a module parameter block, as shown in Figure 5-1. The module parameter block can be subsequently modified by the DDM's configuration routine and thus stores DDM-configurable parameters across resets and power cycles.

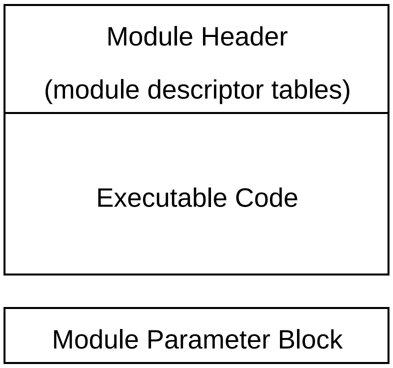


Figure 5-1. DDM Components

This specification protects against installing or updating a driver that corrupts operation. See *Configuration of I/O Device Drivers* in Chapter 2 and section 5.2.5, *Upgrading DDMs*.

5.1.2 Loading DDMs

After the IOP loads its operating environment, it loads and initializes its stored DDMs as necessary. Each DDM is prepended with information (module descriptor tables) that describes the DDM and lists types of adapters and devices that it can control. Using this list, the IOP determines whether the DDM needs to be loaded and assigns it unallocated adapters and devices.

The IOP maintains its own version of a physical resource table that identifies hardware resources (i.e., adapters and controllers) and the DDMs to which they are assigned. Any change in the hardware complement should prompt a configuration dialogue, either to report the loss of service, to acknowledge that a new device or service is available, or to report hardware that is not controlled by a DDM.

5.1.3 DDM Initialization

The IOP calls the DDM's initialization code with a pointer to the module's parameter block as an argument. As the DDM initializes, it registers itself with the IOP (i.e., creates a DDM device). During the registration, the IOP creates an event queue and assigns the DDM a TID.

Messages addressed to that TID are posted to the DDM's event queue. The DDM can now receive DDM class messages.

The IOP permanently assigns a TID to the DDM, either when the DDM is installed or the first time it loads. The IOP maintains the notion of persistent TIDs. That is, each time a DDM is loaded, the IOP assigns it the same TID as the last time it was loaded.

The IOP scans its expansion buses and configuration tables for controllers and adapters that match adapter types specified in the DDM's descriptor tables. For each adapter assigned to the DDM (such a DDM is an HDM), the IOP sends the DDM a ***DdmAdapterAttach*** message identifying the adapter. The DDM initializes the hardware and registers each logical port and/or device with the IOP by creating an I$_2$O device.

As part of the registration, the DDM specifies the TID for each device. The first time a DDM registers a particular device, no TID is assigned. In this case, the IOP assigns a unique TID and reserves it for the DDM. The DDM saves this TID and its relationship to the hardware in its module parameter block, so the DDM can register each device with the same TID each time the DDM is loaded. The IOP must only verify that the TID was reserved for that DDM. It is the DDM's responsibility to maintain TID persistence with the actual hardware.

As the DDM registers devices, it can use its existing event queue, or create additional event queues to handle requests targeted for those devices. The DDM must make this choice before it registers the device.

The IOP contains a configuration database that determines if a registered device is reserved for a local DDM (such a DDM is an ISM). Once a reserved device is registered, the IOP sends a ***DdmDeviceAttach*** message to the ISM indicating the TID of the attached device. The ISM establishes a connection by claiming the device and, just like the HDM, the ISM registers additional devices with the IOP as appropriate.

Each time a DDM registers a device (i.e., created), the IOP updates its logical configuration table. This table identifies services available to the host, other IOPs, and other DDMs (refer to Chapter 4, *Shell Specification*).

5.1.4 Configuration Service

Configuration dialogue is initiated by the host. A DDM requests a configuration dialogue by setting a flag in the IOP's logical configuration table. The host sends the DDM a ***DdmConfigDialog*** request, whose response provides a configuration text for the host to display. As the operator takes appropriate actions, the host sends a ***DdmConfigDialog*** request that returns the results of that operation to the DDM and requests the next dialogue. Each dialogue potentially causes another ***DdmConfigDialog*** request, until the operator exhausts the configuration effort.

The DDM uses the configuration dialogue to set user parameters. The DDM can permanently save these parameters in its *module parameter block*. Except for the header, the content of a module parameter block is solely at the discretion of the DDM.

5.1.5 Message Service

For sending messages, the IOP provides the DDM with an API function call. For receiving messages, the DDM creates one or more event queues. Each event queue is associated with a single thread of execution. When the DDM registers an I_2O device, it specifies which event queue receives messages addressed to that device's TID.

To receive requests, the DDM provides a dispatch table that defines a priority and message handler for each message Function. When the IOP receives a request addressed to that TID, an event is posted to the associated event queue at the priority determined by the Function code. The content of the event specifies the message handler for that message and points to the received message frame.

When a message event reaches the head of the event queue, the message handler is called with a pointer to the message frame.

When creating the reply message, the DDM directly copies the Function, InitiatorAddress, TargetAddress, and InitiatorContext. The DDM may use the message frame of the request for the reply or it may build a new message. This decouples the request from the reply and enables the DDM to hold on to the request until it completes all associated processing. The DDM must explicitly free each received message or post it as a send request.

The IOP handles replies different than requests. Before sending a request, the DDM first identifies the handler for its reply, an event queue, and the priority at which the IOP posts the reply to that event queue. The DDM calls an API that registers these parameters and returns an InitiatorContext value. When the DDM uses that InitiatorContext value in a request, the reply returns with the same value. The IOP retrieves the priority and message handler using that InitiatorContext and posts that reply message to the designated event queue.

Note:

A DDM cannot send requests *to the host. This enforces the practice of the host never receiving unsolicited messages. The host listens to a DDM by posting request messages soliciting replies. When an event occurs, the DDM returns the appropriate reply.*

For peer operation, connection to local devices (I_2O devices residing on the same IOP) is implicit, but a connection must be set up to communicate with an I_2O device on another IOP. The result of the connection setup is that a local TID is assigned as an alias for the remote device. The DDM uses this alias TID to send messages to that module. The use of an alias is transparent to the DDM.

Two mechanisms exist for initiating connections between DDMs.

- **Local configuration:** The IOP contains a configuration database indicating which I_2O devices are permanently assigned to DDMs (as discussed in 5.1.3). In this case, the IOP manages any connection setup, and issues a ***DdmDeviceAttach*** message to the DDM once the device is registered. After issuing ***DdmDeviceAttach*** messages to permanently configured DDMs, the IOP searches module headers and issues a ***DdmDeviceAttach*** message to each DDM that lists itself as a consumer of that class of device.

- **System configuration:** The host sends an ***ExecDeviceAssign*** message to the IOP assigning an I_2O device. The I_2O device can be specifically assigned to a DDM, or it can be generally assigned to the IOP. In the latter case, the IOP determines to which DDM the device is assigned. In addition, the assignment can be temporary or permanent. The IOP places permanent assignments in its configuration database. Just like the local configuration model, the IOP manages any connection setup and issues a ***DdmDeviceAttach*** message to the DDM.

In either case, the ***ExecDeviceRelease*** message revokes the assignment.

The class of a message is determined by the TID in the TargetAddress.

5.1.6 Transport Services: Hardware and Bus Access

The transport services abstract the system bus and any I/O expansion buses hosted by the IOP. Thus, the IOP provides API functions for accessing system resources and adapters. The IOP also provides DMA objects for moving blocks of data between the IOP's memory and system memory, adapters residing on a system bus, or adapters on an I/O expansion bus hosted by the IOP. This provides the mechanism to the DDM for transferring data to and from system memory and for accessing adapters.

The term "system bus" refers to the host processors' main bus and all its I/O buses under the assumption that all components such as main memory and adapter cards have a unique address within that domain.

5.1.6.1 Bus Identification

In the I$_2$O architecture, the IOP is concerned with at least two bus domains: its local bus and the system bus. In addition, there can be a number of I/O expansion buses containing adapters that the IOP accesses. Each of these buses can have its own memory, I/O, and configuration space. Or, that space might be a part of the system address space or the IOP's local address space. Therefore, access to these buses must be abstracted to the DDM by the IOP. On the other hand, a DDM controlling two devices on the same bus may be able to gain performance by understanding that the devices reside on the same bus. Thus, a device's bus identity must be known.

The IOP assigns each bus a handle (busId). The DDM determines the busId for the IOP's local bus and for the system bus via API function calls. When the IOP attaches a physical device or adapter to a DDM, the IOP specifies a handle for the adapter (AdapterId). The DDM uses the AdapterId to identify the adapter and find the busId for its bus. The DDM uses this busId when it accesses the adapter, creates DMA objects, and allocates local memory that is accessible to bus master devices on that bus.

When the DDM allocates memory, it considers which adapters need access to that memory. The levels of access are PRIVATE, SYSTEM, LOCAL_ADAPTERS, ALL_ADAPTERS, and BUS_SPECIFIC. Depending on the IOP's physical configuration and capabilities, many of these levels overlap and may be the same.

- When a DDM allocates memory for internal data structures, it typically specifies PRIVATE memory.
- For data buffers accessible to adapters, the ISM typically specifies ALL_ADAPTER, unless it has specific knowledge of its claimed devices, as well as their devices and adapters.
- The HDM typically specifies LOCAL_ADAPTERS for data buffers, unless it knows all its adapters are on the same bus or it maintains separate instances of buffers for each adapter. In this case, the HDM might specify BUS_SPECIFIC.

- A management ISM that does not generate base class messages (i.e., no data flow to adapters) typically specifies PRIVATE. But if it needs to share its data structures with a peer entity on another IOP, it specifies SYSTEM access.

Cache coherency is another concern when allocating memory. Many I/O class processors do not support snooping protocols, and even when they exist, caching I/O data buffers can lead to undesirable effects. Typically, I/O data buffers are allocated in uncached memory and private memory is always cached.

The DDM allocates data buffers accessible by a bus master adapter on an I/O bus. However, there is an offset between the local address and the bus address. The DDM converts a local memory address to a bus address, or vice versa, by calling the address translation function and specifying the appropriate busIds.

5.1.6.2 Address Translation

When using shared memory, it should be noted that a shared memory location has three addresses: one that identifies its position in the system memory map, one that identifies its position in the IOP's local memory map, and possibly one that identifies its position in the IOP's expansion bus memory map.

Address translation is necessary because:

1. The only common memory space among IOPs is system memory. Therefore, the only appropriate reference to a shared memory location when communicating with a remote DDM is its system memory address.
2. A bus master adapter resides in the address space of the expansion bus. Therefore, the DDM must use the expansion bus address instead of the local address when programming the adapter's DMA engine to read or write to the adapter's memory or I/O ports.

These needs require that a DDM have a translation mechanism that can:

1. convert a local memory address to a system address, and vice versa.
2. convert a local memory address to an expansion bus address, and vice versa.

Given a system address, a local address, or an expansion bus address of a location in a shared memory partition, the DDM can translate it to a system, local, or an expansion bus address. This is accomplished via the API translation function call. The DDM specifies the bus, the address on that bus, and a second bus that needs to access that memory location. The function returns the address for that second bus.

5.1.6.3 Transport Functions

Because access to different buses and spaces on those buses may vary between platforms, the DDM does not directly access system memory or adapters on expansion buses. Instead, two sets of functions access system memory and adapters. One set performs block transfers, and the other performs single accesses.

For block transfers, the DDM creates DMA objects. The DMA object is used to move a block of data between IOP local memory and either an adapter or system memory. The DMA action queues the request to the IOP's hardware and returns control to the calling module before the

DMA completes. After the DMA is complete, the IOP posts a DMA completion event to an event queue specified by the DDM.

For single accesses, a set of functions provides a more direct mechanism for reading and writing memory, I/O ports, or configuration registers for a particular bus. These functions operate on single access data (i.e., a byte, 16-bit word, 32-bit word, or 64-bit word) and the function is immediate; that is, the call returns only once the transfer completes. This causes an immediate bus access independent of the DMA, but it may be delayed until the current DMA completes. Therefore, these direct functions are effective only for accessing small amounts of data.

5.1.7 OS Services

DDMs execute on the IOP under IRTOS, a special-purpose real-time OS designed specifically to support high-speed, low-overhead I/O operations.

System vendors that provide open I/O processors that host arbitrary device drivers must provide the IRTOS environment. Device vendors that provide the drivers to run on those I/O processors must create DDMs that conform to the IRTOS environment.

IRTOS is organized around two primary concepts: objects and events. The IRTOS driver abstraction is based on an event-driven model of device drivers. For more on IRTOS, see Section 5.3.3.

5.2 Principles of Operation

This section discusses basic IOP operations, including installing, configuring, and initializing DDMs, message queuing, and transport services.

5.2.1 Installing DDMs

DDMs are installed because the host sends a ***ExecSwDownload*** message. This message provides the module code and an optional module parameter block. The module parameter block is analogous to a *.INI file and stores parameters governing DDM operation. Installing a DDM does not automatically cause the DDM to load. The DDM itself is prepended with a set of descriptor tables that identifies the DDM and gives the IOP a list of adapters and device types that the DDM can control. Based on this list, the IOP determines if a DDM needs to be loaded and assigns it adapters and devices.

5.2.2 DDM Initialization

After a DDM is loaded, its main initialization entry point is invoked with a pointer to the module's parameter block. The module's initialization code initializes the module, creates a DDM object that sets up an event queue for the DDM, and registers the module as a DDM class I$_2$O device (i.e., **i2oDdmCreate()** function). The IOP responds to the registration by assigning a TID to the DDM.

The IOP searches its physical configuration database for any available adapters assigned to the DDM. The IOP matches the adapter's signature (e.g., information from the adapter's configuration registers) against the adapter signatures in the module's header. For each match, the IOP posts a ***DdmAdapterAttach*** message to the DDM's event queue. The DDM initializes the adapter and registers as many I_2O devices as necessary to represent the services available (i.e., the **i2oDevCreate()** function). As an example, a SCSI DDM would create a bus adapter class I_2O device for each SCSI port, and for each SCSI device, create a SCSI peripheral class I_2O device. The DDM uses information in its module parameter block to register devices consistently with the same TID.

Each registered I_2O device is assigned a TID. The DDM can create additional event queues if necessary and, when it creates an I_2O device, it identifies which event queue processes messages addressed to that device's TID.

As I_2O devices are registered, the IOP checks its database and determines if the I_2O device is reserved for a local ISM. If it is, a ***DdmDeviceAttach*** message is posted to that DDM's event queue. Additionally a ***DdmDeviceAttach*** message is posted when the host sends an ***ExecDeviceAssign*** message to the IOP. Instead of initializing hardware, as with the ***DdmAdapterAttach*** , the ISM sends an ***UtilClaim*** message to the attached I_2O device. Like the HDM, when an ISM processes an ***DdmDeviceAttach*** message, it can register (create) additional I_2O devices.

An I_2O device is considered sharable (i.e., the service supports more than one user) but can limit the number of users it can support. That limit can be one or more. An ISM or OSM sends the ***UtilClaim*** message to the I_2O device to use the I_2O device's base class resources (i.e., become the primary, alternate, or secondary user). Whether or not a DDM reaches its user limit, it must receive and respond to utility class messages from all DDMs (***UtilClaim*** is a utility message). If the I_2O device has not reached its user limit, it accepts ***UtilClaim*** requests; otherwise, it rejects subsequent ***UtilClaim*** requests. Only one primary user may exist at a time but the primary user may allow alternate authorized users and peer service users (see Chapter 6, *Utility Messages*). A user sharing rights can use a device's services exclusively for temporary periods via the ***UtilLock*** and ***UtilLockRelease*** messages. The driver acquires general rights to a device that may be shared between different DDMs, IOPs, or units via the ***UtilDeviceReserve*** message.

When the I_2O device is initially created, an entry is placed in the IOP's logical configuration table with the UserTID field set to unknown. When a primary user claims the device, the DDM updates the table's UserTID to reflect the TID of its primary user. This signifies that the resource is not available.

5.2.3 IOP Initialization Example

To illustrate the process, consider the configuration shown in Figure 5-2, which has two SCSI adapters and a number of SCSI devices attached to each port. Assume that a single HDM controls both SCSI ports and that there are three separate ISMs. One ISM is a general block storage module configured to control all the disks on both SCSI ports. The second ISM is a RAID (block storage) module configured to control the first five disks on the first SCSI port. The third ISM controls the tape drive on the first SCSI port. The scanner is not an I_2O class and is therefore left for an OSM to control as a typical SCSI device.

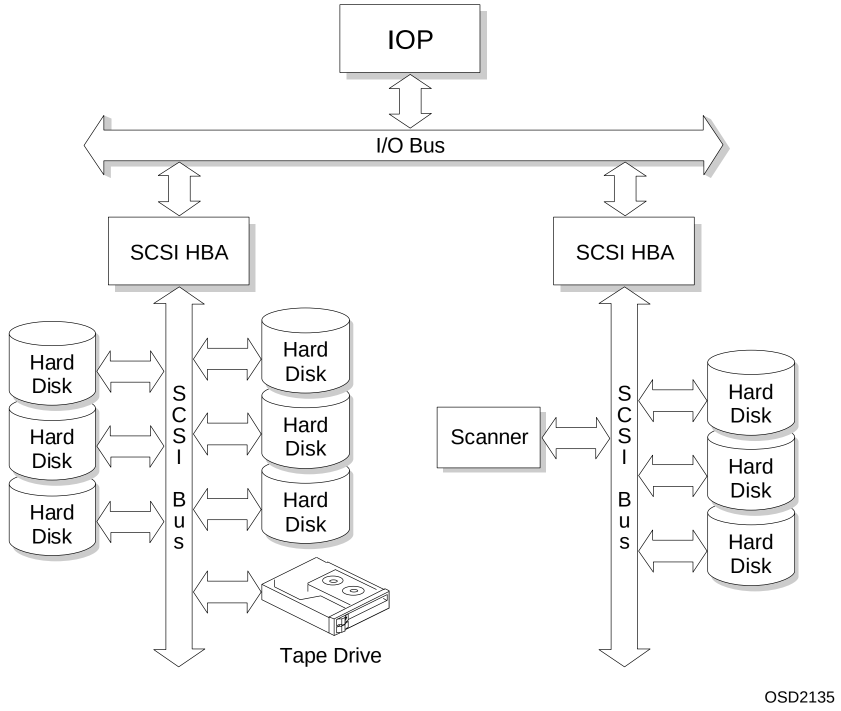


Figure 5-2. An IOP Physical Configuration

A typical initialization sequence follows:

1. By definition, the IOP executive is always TID 000h, and the 001h TID is reserved for host OSMs.
2. The IOP discovers the SCSI adapters, and thus loads and initializes the SCSI HDM. As part of its initialization procedure, the HDM registers itself as a DDM and is assigned a TID of 008h.

The next phase is hardware initialization:

3. The IOP posts a ***DdmAdapterAttach*** message to the SCSI HDM's queue, which contains the AdapterID for the *first* SCSI port.
4. The HDM creates a bus adapter (SCSI port) I$_2$O device, specifying TID = 010h.
5. As the SCSI HDM discovers the six disk drives and the tape drive, it creates appropriate SCSI peripheral class I$_2$O devices, specifying them as TID = 011h through 017h.

The logical configuration table now contains the information as shown in Table 5-1.

Table 5-1. Initial Configuration Information

TID	Class	SubClass	Parent	UserTID	Claimed Devices	Notes
008h	DDM	HDM	000h	000h		{SCSI driver}
010h	Bus Adapter	SCSI	008h	FFFh		
011h	SCSI Peripheral	Hard Disk	010h	FFFh		
012h	SCSI Peripheral	Hard Disk	010h	FFFh		
013h	SCSI Peripheral	Hard Disk	010h	FFFh		
014h	SCSI Peripheral	Hard Disk	010h	FFFh		
015h	SCSI Peripheral	Hard Disk	010h	FFFh		
016h	SCSI Peripheral	Hard Disk	010h	FFFh		
017h	SCSI Peripheral	Tape Drive	010h	FFFh		

9. As a result of the SCSI Peripheral devices being registered, the IOP loads and initializes the general block storage ISM. The ISM registers itself as a DDM and is assigned TID = 009h.

10. Next, the IOP posts a ***DdmDeviceAttach*** message to the generic block storage ISM for each matching SCSI Peripheral (see Table 5-12: Device Table declarations, part of Module Descriptor Header). Since the ISM remembers the correlation between the devices it claims and the devices it creates, for each attached device it recognizes, the ISM creates a block storage class device using the remembered TID, (TIDs = 21h through 26h). Each block storage device then claims the corresponding SCSI peripheral block storage device.

11. Also as a result of the SCSI Peripheral devices being registered, the IOP loads and initializes the tape storage ISM. The ISM registers itself as a DDM and it is assigned TID = 00Ah.

12. The IOP now posts a ***DdmDeviceAttach*** message to the tape ISM for each matching SCSI Peripheral. The tape ISM recognizes TID 017, creates a tape storage class device (tape drive), specifying TID = 027h, and then the tape device (TID 027) claims TID 017.

13. As a result of the block storage class devices being registered, the IOP loads and initializes the RAID ISM. The RAID ISM registers itself as a DDM and is assigned TID = 00Bh.

14. Next, the IOP posts a ***DdmDeviceAttach*** message to the RAID ISM for each block storage device registered (TIDs 21h through 26h). The RAID ISM recognizes the first TID attached and creates a block storage class device (RAID disk), which is TID = 028h. The new RAID disk claims the appropriate devices (TIDs 21h through 25h).

    Note: The RAID device might determine which devices to claim either from information stored in its MPB or by inspecting the drives themselves.

    The logical configuration table now contains the information as shown in Table 5-2.

Table 5-2. Configuration Information after First Adapter Initialization

TID	Class	SubClass	Parent	UserTID	Claimed Devices	Notes
008h	DDM	HDM	000h	000h		{SCSI driver}
009h	DDM	ISM	000h	000h		{Block Storage}
00Ah	DDM	ISM	000h	000h		{Tape driver}
00Bh	DDM	ISM	000h	000h		{RAID driver}
010h	Bus Adapter	SCSI	008h	FFFh		
011h	SCSI Peripheral	Hard Disk	010h	021h		
012h	SCSI Peripheral	Hard Disk	010h	022h		
013h	SCSI Peripheral	Hard Disk	010h	023h		
014h	SCSI Peripheral	Hard Disk	010h	024h		
015h	SCSI Peripheral	Hard Disk	010h	025h		
016h	SCSI Peripheral	Hard Disk	010h	026h		
017h	SCSI Peripheral	Tape Drive	010h	027h		
021h	Block Storage	Disk	009h	028h	011	
022h	Block Storage	Disk	009h	028h	012	
023h	Block Storage	Disk	009h	028h	013	
024h	Block Storage	Disk	009h	028h	014	
025h	Block Storage	Disk	009h	028h	015	
026h	Block Storage	Disk	009h	FFFh	016	
028h	Block Storage	RAID Disk	00Bh	FFFh	021,022,023,024,025	
027h	Tape Storage	DAT	00Ah	FFFh	017	

Shaded area signifies data that is unchanged from the previous example.

1. The IOP posts another ***DdmAdapterAttach*** message to the SCSI HDM with the AdapterID for the *second* SCSI port.

2. The HDM creates another bus adapter class I$_2$O device, specifying TID = 030h.

3. As the SCSI HDM discovers the three disk drives and the scanner on the second port, it creates appropriate SCSI peripheral devices registering them as TID = 031h through 034h.

4. The IOP posts an ***DdmDeviceAttach*** message to the generic block storage ISM for each SCSI peripheral device registered that matches the ISM's device Table (TIDs 031 through 034h). Remembering the association, the ISM creates a block storage class device (disk drive), specifying remembered TIDs 041h, 042h, and 043h. These storage class devices claim the appropriate SCSI peripheral devices.

The logical configuration table now contains the information as shown in Table 5-3:

Table 5-3. Final Logical Configuration Table

TID	Class	SubClass	Parent	UserTID	Claimed Devices	Notes
008h	DDM	HDM	000h	000h		{SCSI driver}
009h	DDM	ISM	000h	000h		{Block Storage}
00Ah	DDM	ISM	000h	000h		{Tape driver}
00Bh	DDM	ISM	000h	000h		{RAID driver}
010h	Bus Adapter	SCSI	008h	FFFh		
030h	Bus Adapter	SCSI	008h	FFFh		
011h	SCSI Peripheral	Hard Disk	010h	021h		
012h	SCSI Peripheral	Hard Disk	010h	022h		
013h	SCSI Peripheral	Hard Disk	010h	023h		
014h	SCSI Peripheral	Hard Disk	010h	024h		
015h	SCSI Peripheral	Hard Disk	010h	025h		
016h	SCSI Peripheral	Hard Disk	010h	026h		
017h	SCSI Peripheral	Tape Drive	010h	027h		
031h	SCSI Peripheral	Hard Disk	030h	041h		
032h	SCSI Peripheral	Hard Disk	030h	042h		
033h	SCSI Peripheral	Hard Disk	030h	043h		
034h	SCSI Peripheral	unknown	030h	FFFh		scanner
021h	Block Storage	Disk	009h	028h	011	
022h	Block Storage	Disk	009h	028h	012	
023h	Block Storage	Disk	009h	028h	013	
024h	Block Storage	Disk	009h	028h	014	
025h	Block Storage	Disk	009h	028h	015	
026h	Block Storage	Disk	009h	FFFh	016	
028h	Block Storage	RAID Disk	00Bh	FFFh	021,022,023,024,025	
041h	Block Storage	Disk	009h	FFFh	031	
042h	Block Storage	Disk	009h	FFFh	032	
043h	Block Storage	Disk	009h	FFFh	033	
027h	Tape Storage	DAT	00Ah	FFFh	017	

Shaded area signifies data that is unchanged from the previous example.

When the host reads the logical configuration table, entries with a value other than FFFh in the UserTID field are reserved and not available to an OSM.

When the OSM (or an ISM) claims a device, the UserTID field is updated to indicate that the device is no longer available to others.

Figure 5-3 illustrates the logical configuration of the above example. The I$_2$O devices above the modules are available to an OSM, or an ISM on another IOP. Reserved devices are shown below the relative user.

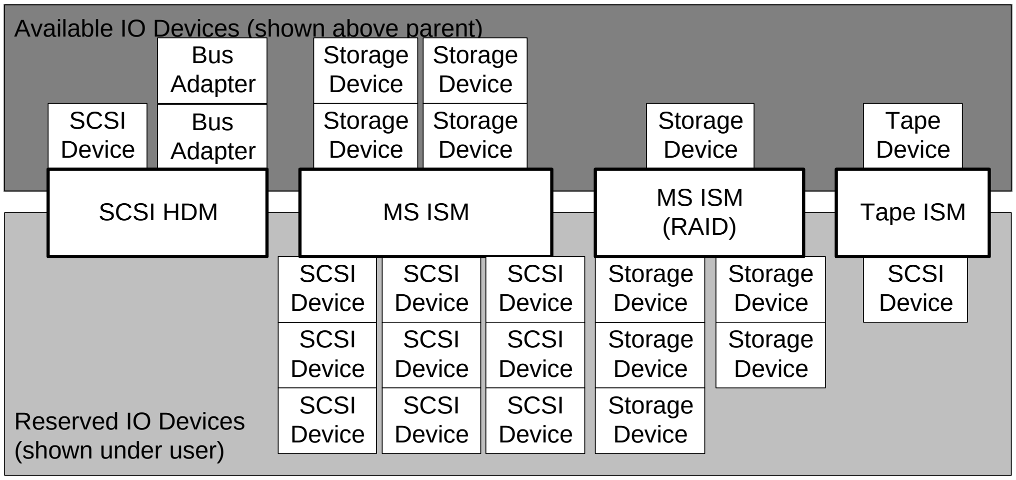


Figure 5-3. An IOP's Logical Configuration

5.2.4 DDM Configuration

When a DDM is installed, its module header lists the adapter types that the DDM can control. The IOP compares those types against its own list of unresolved resources. For any match, the IOP can invoke a configuration dialogue with the host/user to confirm the adapter's assignment to the DDM. Typically the IOP does not wait for the configuration dialogue, but rather configures adapters according its own set of rules, issues the ***DdmAdapterAttach,*** and requests the configuration dialogue. This allows confirmation or modification of the configuration after the system is operational.

When the IOP issues the ***DdmAdapterAttach*** message to the DDM, the DDM creates I_2O devices related to that hardware. Because only the DDM understands the relationship of the hardware to the registered device, the DDM must correlate the TID assignment of I_2O devices it registers. The mechanism for this task lies in the module parameter block. The first time an adapter is assigned to a DDM, the DDM specifies a TID of 000h for each new I_2O device it creates. The IOP, detecting this special value, assigns the I_2O device an unique TID. The DDM stores the relationship of that new TID to the hardware in its module parameter block. Thus, the next time the IOP is booted and the adapter is attached to the DDM, the DDM creates the I_2O device specifying the same TID as previously assigned. The IOP simply verifies that the TID is assigned to that DDM. This assures that devices are assigned TIDs consistently and that the host can use the TID assignment to recognize new or missing devices. The DDM must determine whether an adapter or device is the same, a replacement, or considered a new device. The DDM may use the configuration dialogue for this effort.

DdmAdapterAttach events are not guaranteed to occur in the same order each time the IOP is reset. The adapter may not be in the same location, nor have the same AdapterID. Therefore, the best correlation to hardware is a manufacturer's serial number, MAC address, or other unique identification that moves with a device. With this level of correlation, the DDM can expressly track hardware when it is moved to a different slot or replaced. The next best correlation is the hardware's association with its physical bus location.

The DDM must assure that it does not assign a TID to a different entity. If the DDM detects a hardware change, it should defer I_2O device creation and use a configuration dialogue to confirm whether the new card is a replacement or a new device. A replacement should receive the same TID, whereas a new device requires a new TID. On the other hand, if the adapter controls the boot device, it may be inappropriate to wait for a configuration dialogue.

If the DDM requires a configuration dialogue, it sets a flag to request a dialogue. The host invokes the configuration dialogue at its convenience.

5.2.5 Upgrading DDMs

The IOP can receive a request to install a DDM that upgrades a DDM already installed and running. This version of the specification does not require hot swapping of DDMs but, protects against upgrading to a faulty DDM. See *Configuration of I/O Device Drivers* in Chapter 2.

If the suspect DDMs are rejected implicitly, because the user has neither accepted nor rejected them, the IOP must request a configuration dialogue to warn the user that the experimental DDMs are not loaded. The host must issue a new installation command if it wants to try the upgrade again.

Until suspect DDMs are either accepted or rejected, the IOP should not accept another driver installation request.

5.2.6 Message Queuing

Each I_2O device is created with a unique TID and associated with an event queue. The event queue supports eight levels of priority, and the DDM supplies a pointer to the message dispatch table when the device is created. This table lists Function codes, their priorities and message handlers for processing request messages. When the IOP receives a request message to that TID, the IOP uses the priority value corresponding with the message's Function code and queues the *request* to the event queue. A DDM can either point to the same dispatch table for all its registered devices or supply a different table for each.

If the DDM sends requests, it must register their reply handlers and priorities via an API function call. The IRTOS places that information in a structure and returns an InitiatorContext value identifying that structure. When a DDM sends a request, it uses the appropriate InitiatorContext value. When a reply is received, the IRTOS retrieves the reply handler and priority from a structure identified by the InitiatorContext field and then queues the event.

5.2.7 Accessing System Memory

As mentioned earlier, a DDM cannot access system memory directly, but must use the IOP's transport services. The transport services provide DMA capability for moving blocks of data between system memory and a local buffer. The IOP's transport services also support single accesses. The single access is simpler, because it does not involve a callback mechanism. However, the DMA is more efficient (for both the local and system buses) when transferring more than a few bytes of data. See section 5.1.6.3 for more information.

5.2.8 Accessing Physical Adapters

A DDM cannot access physical adapters directly, but must use the IOP's transport services. Access to physical adapters is the same as specified for system memory access (see 5.2.7).

An adapter can directly access a portion of IOP memory. This enables bus master adapters to copy data directly to the DDM's buffers. The DDM must allocate buffers in a memory partition accessible by the adapter. See section 5.1.6.2 for more information.

5.3 Technical Reference

This section is the technical reference for the DDM class of device. It defines the messages the IOP uses to manage DDMs, the physical structure of the DDM, and its module parameter block.

5.3.1 DDM Class Message Definitions

The following messages are specifically designed for managing loadable DDMs:

Table 5-4. DDM Class Messages for Loadable DDMs

Mnemonic	Description
DdmAdapterAttach	Assigns a physical adapter to a HDM
DdmAdapterReconfig	Notifies HDM of adapter's new physical location and resumes operation
DdmAdapterRelease	Revokes a physical adapter previously assigned to a HDM
DdmAdapterResume	Resumes access after a ***DdmAdapterSuspend*** when no change occurred
DdmAdapterSuspend	Suspends access to a adapter so it can be reconfigured
DdmDeviceAttach	Assigns an I$_2$O device to an ISM (forms connection)
DdmDeviceRelease	Revokes a device previously assigned to an ISM
DdmDeviceReset	Aborts all operation with the specified TID
DdmDeviceResume	Resumes sending messages to the specified TID
DdmDeviceSuspend	Suspends sending messages to the specified TID
DdmSelfReset	Aborts all operations and reset state
DdmSelfResume	Resumes sending messages
DdmSelfSuspend	Suspends operation of the module
DdmSystemChange	Notifies DDM that system has been reconfigured
DdmSystemEnable	Notifies DDM that system is operating after a ***DdmSystemHalt***
DdmSystemHalt	Notifies DDM that system will be reconfigured

All DDM class messages are single-transaction messages. Typically, the MessageFlags field for requests should be set to 00h (for 32-bit context size) or 02h (for 64-bit context size). For a normal reply, it should contain C0h (for 32-bit context size) or C2h (for 64-bit context size). All requests are single transaction requests. All replies are single transaction replies. Since no request provides an SGL, the VersionOffset field should be 01h for both requests and replies.

Detailed Status Codes for replies are specified in Chapter 3.

5.3.1.1 DdmAdapterAttach

This message gives the DDM the identity of an adapter and requests that the DDM accept control and initialize the adapter. The IOP should defer the reply to this message until it has registered all devices created as a result of this message.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmAdapterAttach	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
AdapterID				16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 5-4. *DdmAdapterAttach* Request Message Structure

Fields

AdapterID Handle for the adapter. The DDM and IOP use this value to represent the physical adapter in subsequent messages and function calls.

MessageFlags Typically 00h for 32-bit context size and 02h for 64-bit context size.

5.3.1.2 DdmAdapterReconfig

This message returns adapter control to the DDM after a ***DdmAdapterSuspend*** operation and notifies the DDM that the adapter is physically reconfigured. The DDM should ascertain the adapter's new configuration.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmAdapterReconfig	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
AdapterID				16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 5-5. *DdmAdapterReconfig* Request Message Structure

5.3.1.3 DdmAdapterRelease

This message revokes access to an adapter and asks the DDM to release control of it. The DDM should put the adapter in a quiescent state before replying to this message.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmAdapterRelease	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
AdapterID				16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 5-6. *DdmAdapterRelease* Request Message Structure

5.3.1.4 DdmAdapterResume

This message returns control of the adapter to the DDM after a ***DdmAdapterSuspend*** operation and notifies the DDM that the adapter was not physically reconfigured.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmAdapterResume	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
AdapterID				16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 5-7. *DdmAdapterResume* Request Message Structure

5.3.1.5 DdmAdapterSuspend

This message temporarily revokes access to an adapter. The IOP sends this message before changing the adapter's physical address. Once the DDM replies to this message, it should not access the adapter until it receives a ***DdmAdapterReconfig*** or ***DdmAdapterResume*** message.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmAdapterSuspend	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
AdapterID				16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 5-8. *DdmAdapterSuspend* Request Message Structure

5.3.1.6 DdmDeviceAttach

This message provides the DDM with the TID of an I$_2$O device and suggests that the DDM claim and use the device. This is how the IRTOS assigns devices to ISMs. The IOP should defer the reply to this message until it has registered all devices created as a result of this message.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmDeviceAttach	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
reserved			DeviceTID	16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 5-9. *DdmDeviceAttach* Request Message Structure

5.3.1.7 DdmDeviceRelease

This message revokes access to an I$_2$O device previously attached to the DDM and requests that the DDM release control of the device. The DDM should send an ***UtilClaimRelease*** message to the device.

<table>
<tr><th colspan="2">31 3 24</th><th colspan="2">23 2 16</th><th colspan="2">15 1 8</th><th colspan="2">7 0 0</th><th></th></tr>
<tr><td colspan="4">MessageSize</td><td colspan="2">MessageFlags</td><td colspan="2">VersionOffset = 01h</td><td>0</td></tr>
<tr><td colspan="2"><b><i>DdmDeviceRelease</i></b></td><td colspan="3">InitiatorAddress</td><td colspan="3">TargetAddress</td><td>4</td></tr>
<tr><td colspan="8">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="8">TransactionContext</td><td>12 (16)</td></tr>
<tr><td colspan="5">reserved</td><td colspan="3">DeviceTID</td><td>16 (24)</td></tr>
</table>

Offset in () signifies offset for 64-bit context fields

Figure 5-10. *DdmDeviceRelease* Request Message Structure

5.3.1.8 DdmDeviceReset

The DDM aborts all operations for the specified TID. If the DDM is the parent of the ResetTID, then it aborts all request messages queued for that TID before replying to this message. It also aborts outstanding I/O requests using the appropriate process abort status code. Otherwise, the DDM flushes all messages and outstanding I/O requests from the specified TID. The DDM does not reply to the ResetTID after receiving this message. In either case, the claim state reverts to *not claimed*.

The typical order of messages is:

1. ***DdmDeviceSuspend*** to the users of the TID to reset
2. ***DdmDeviceReset*** to the device being reset
3. ***DdmDeviceReset*** to each of its users

<table>
<tr><th colspan="2">31 3 24</th><th colspan="2">23 2 16</th><th colspan="2">15 1 8</th><th colspan="2">7 0 0</th><th></th></tr>
<tr><td colspan="4">MessageSize</td><td colspan="2">MessageFlags</td><td colspan="2">VersionOffset = 01h</td><td>0</td></tr>
<tr><td colspan="2"><b><i>DdmDeviceReset</i></b></td><td colspan="3">InitiatorAddress</td><td colspan="3">TargetAddress</td><td>4</td></tr>
<tr><td colspan="8">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="8">TransactionContext</td><td>12 (16)</td></tr>
<tr><td colspan="5">reserved</td><td colspan="3">ResetTID</td><td>16 (24)</td></tr>
</table>

Offset in () signifies offset for 64-bit context fields

Figure 5-11. *DdmDeviceReset* Request Message Structure

5.3.1.9 DdmDeviceResume

This message enables the DDM to send messages to and from the specified TID again. If the DDM is the parent of the SuspendedTID, it enables processing request messages for that TID. Otherwise, it enables sending requests or replies to that TID.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmDeviceResume	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
reserved		SuspendedTID		16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 5-12. *DdmDeviceResume* Request Message Structure

5.3.1.10 DdmDeviceSuspend

The DDM suspends message service with the specified TID until it receives a ***DdmDeviceReset***, ***DdmDeviceResume***, or ***DdmSelfReset*** message. If the DDM is the parent of the SuspendedTID, then it stops processing request messages for that TID. Otherwise, it stops sending requests or replies to that TID. The typical use of this message is to halt communication while the system is reconfigured.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmDeviceSuspend	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
reserved		SuspendedTID		16 (24)

Offset in () signifies offset for 64-bit context fields

Figure 5-13. *DdmDeviceSuspend* Request Message Structure

5.3.1.11 DdmSelfReset

The DDM discards all operations and resets its state.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmSelfReset	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)

Offset in () signifies offset for 64-bit context fields

Figure 5-14. *DdmSelfReset* Request Message Structure

When it receives this message, the target aborts all operations, releases all devices and other IRTOS objects (except for the DDM object itself), and returns all message frames before replying. The DDM must destroy all I$_2$O devices it created and waits for the IOP to reissue ***DdmAdapterAttach*** and ***DdmDeviceAttach*** messages.

5.3.1.12 DdmSelfResume

When it receives this message, the DDM resumes operation and begins sending messages again.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmSelfResume	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)

Offset in () signifies offset for 64-bit context fields

Figure 5-15. *DdmSelfResume* Request Message Structure

5.3.1.13 DdmSelfSuspend

This message asks the target to suspend operation of the module and stop sending messages. Once the target responds to this message, it sends no messages until it receives a ***DdmSelfReset*** or ***DdmSelfResume*** message.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmSelfSuspend	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)

Offset in () signifies offset for 64-bit context fields

Figure 5-16. *DdmSelfSuspend* Request Message Structure

5.3.1.14 DdmSystemChange

This message notifies the DDM of system configuration changes, and that it can resume operation. If the DDM is attached to any adapters, it should ascertain each adapter's new configuration. If the DDM has any direct links to DDMs on other IOPs, it should re-establish those links and validate any system addresses in use.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmSystemChange	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)

Offset in () signifies offset for 64-bit context fields

Figure 5-17. *DdmSystemChange* Request Message Structure

5.3.1.15 DdmSystemEnable

This message notifies the DDM that the system configuration has not changed, and the DDM can resume operation.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmSystemEnable	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)

Offset in () signifies offset for 64-bit context fields

Figure 5-18. *DdmSystemEnable* Request Message Structure

5.3.1.16 DdmSystemHalt

This message notifies the DDM that the system is about to be reconfigured. Once the target responds, it should not access any adapters on a system bus until it receives a ***DdmSystemChange*** or a ***DdmSystemEnable*** message.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
DdmSystemHalt	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)

Offset in () signifies offset for 64-bit context fields

Figure 5-19. *DdmSystemHalt* Request Message Structure

5.3.2 Technical Reference for Structures

This section describes the physical structure of the DDM and its module parameter block.

5.3.2.1 DDM Physical Structure

The DDM contains a module descriptor header (set of module descriptor tables) prepended to the executable code, as specified in the following tables.

Table 5-5 Structure of Module Descriptor Header

Size	Name	Description
32 bits	HeaderSize	Number of bytes consumed by module descriptor header, starting with this field and including all descriptor tables. Always an multiple of four bytes.
16 bits	OrganizationID	The I_2O organization ID assigned to the vendor producing the module.
16 bits	ModuleID	A value assigned by the vendor producing the module. The OrganizationID and ModuleID form a unique name for the module, which identifies it during upgrades.
16 bits	DateCodeDay	ASCII string identifying the two-digit day of the month the DDM was produced (01 through 31).
16 bits	DateCodeMonth	ASCII string identifying the two-digit month the DDM was produced (01 through 12).
32 bits	DateCodeYear	ASCII string identifying the four-digit year the DDM was produced.
8 bits	I2OVersionInfo	This version is 01h.
8 bits	MajorCapabilities	This field identifies major capabilities, specified by Table 5-6.
16 bits	reserved	Set to zero. Reserved for future definition.
32 bits	ObjCodeSize	Number of bytes of memory required to load and execute the code.
32 bits	TableOffset	Offset from the start of the module header (i.e., HeaderSize field) to the start of the tables (i.e., NumberTables field). This allows additional fields in the future.
32 bits	MemoryReqmnts	Number of bytes of data memory needed before attaching any adapters or devices.
32 bits	MemoryPreferred	The additional amount of memory (in bytes) of IOP memory desired by the module for enhanced performance. This field excludes the amount specified by the MemoryReqmnts field.
32 bits	ModuleVersion	Module version (four ASCII characters).
8 bits	ProcessorType	Type of IOP: 00h Intel 80960 series 01h AMD29000 series 02h Motorola 68000 series 03h ARM series 04h MIPS series 05h Sparc series 06h PowerPC series 07h Alpha series 08h Intel x86 series FFh Other Other values are reserved (for current list see I2O SIG Web site at http://www.i2osig.org/).
8 bits	ProcessorVersion	Version of processor dependent on ProcessorType: 00h default other values to be determined (for current list visit the I2O SIG Web site at http://www.i2osig.org/).

8 bits	ObjCodeFormat	Identifies how the object code was compiled, dependent on ProcessorType: 00h default other values to be determined (for current list visit the I2O SIG Web site at http://www.i2osig.org/).
8 bits	reserved1	reserved
x32 bits	unknown	Additional fields defined in future versions. Since size of additional fields are unknown, use TableOffset to locate following tables.
32 bits	NumberTables	Number of descriptor tables.
24 bytes	ModuleInfo	ASCII string; module name and any manufacturer's information the vendor provides
16 bits	Length	Length of this table in 32-bit words (includes this field).
16 bits	DescriptorID	Table content, per **Table 5-7**. Tables are in order of their DescriptorID.
x32 bits	*data*	Table content
x32 bits	*additional tables*	Length, DescriptorID, and *data*, as above, repeated for each table. Tables are in order of their DescriptorID.
any	*object code*	Object code; HeaderSize determines placement of object code.

x32 bits means any number of 32-bit words.

Table 5-6 Module's Major Capabilities

Bit Position	Description
Bit 1,0	ContextFieldSizeCapability 0,0 Supports only 32-bit context fields. 0,1 Supports only 64-bit context fields. 1,0 Supports both 32-bit or 64-bit context fields, but not concurrently. 1,1 Supports 32-bit & 64-bit context fields concurrently.
others	reserved

Additional capabilities may be specified at a later date.

Table 5-7 DescriptorID Assignments

DescriptorID	Description
0000h	**Index table** – indicates the tables included in the header; see Table 5-8
0001h	**Adapter table** – describes the adapters that this HDM can control; see Table 5-10
0002h	**Device table** – list of device classes this ISM can control; see Table 5-12
0003h	**Obsolete DDM table** – list of ModuleIDs that this module replaces; see Table 5-14
0004h	**TCL script table** – table of TCL Script for Configuration Dialogue; see Table 5-16.

Additional DescriptorIDs will be specified.

Table 5-8 Structure of Index Table

Size	Name	Description
16 bits	length	Length, in 32-bit words, of table, including this field
16 bits	DescriptorID	DescriptorID = 0000h identifies this as an Index table
8 bits	EntrySize	Number of 32-bit words in each entry (two for this version)
8 bits	NumberEntries	Number of entries
8 bits	TableVersion	This version = 00h
8 bits	reserved1	reserved
n x m	*data*	Table entries (size = EntrySize x NumberEntries x 32 bits); see Table 5-9

Table 5-9 Index Table Entry

Size	Name	Description
16 bits	TableDescriptorID	DescriptorID of the Table
16 bits	reserved2	reserved
32 bits	TableOffset	Offset of table from start of header

Table 5-10 Structure of Adapter Table

Size	Name	Description
16 bits	length	Length, in 32-bit words, of table, including this field
16 bits	DescriptorID	DescriptorID = 0001h identifies this as an adapter table
8 bits	EntrySize	Number of 32-bit words in each entry
8 bits	NumberEntries	Number of entries
8 bits	TableVersion	This version = 00h
8 bits	-	reserved
n x m	*data*	Table entries (size = EntrySize x NumberEntries x 32 bits). The first byte of each entry identifies the bus type; the remainder is specific to bus type. PCI is the only type defined (see Table 5-11).

Table 5-11 PCI Adapter Table Entry

Size	Name	Description
8 bits	BusType=PCI	See Chapter 3 for bus type values.
24 bits	ClassCode	Value to match PCI configuration register at offset 09h
16 bits	PCIvendorID	Value to match PCI configuration register at offset 00h
16 bits	PCIdeviceID	Value to match PCI configuration register at offset 02h
16 bits	PCIdeviceMask	Value ANDed with the PCI configuration register at offset 02h, before comparing with PCIdeviceID
16 bits	SubVendorID	Value to match PCI configuration register at offset 2Ch
16 bits	SubDeviceID	Value to match PCI configuration register at offset 2Eh
16 bits	SubDeviceMask	Value ANDed with the PCI configuration register at offset 2Eh before comparing with SubDeviceID
32 bits	MemReqmnts	Number of bytes of IOP data memory needed to support this adapter

Table 5-12 Structure of Device Table

Size	Name	Description
16 bits	length	Length, in 32-bit words of table, including this field
16 bits	DescriptorID	DescriptorID = 0002h identifies this as an device table.
8 bits	EntrySize	Number of 32-bit words in each entry (two for this version)
8 bits	NumberEntries	Number of entries
8 bits	TableVersion	This version = 00h
8 bits	-	reserved
n x m	*data*	Table entries (size = EntrySize x NumberEntries x 32 bits) First two words of each entry identify the message class and subclass. The IRTOS determines if an I_2O device can be assigned to this ISM by matching the ClassID and SubClass with the same fields in logical configuration table entries. See Table 5-13

Table 5-13 Device Table Entry

Size	Name	Description
32	ClassID	The first word of each entry identifies the message class (see *Class Codes* in Chapter 6). A value of -1 means issue a ***DdmDeviceAttach*** message for all devices as they are registered. A value of 0 means always load and initialize this module.
32	SubClass	The second word of each entry identifies the subclass (defined by each message class in Chapter 6). A value of -1 indicates any subclass.

Table 5-14 Structure of Obsolete DDM Table

Size	Name	Description
16 bits	length	Length, in 32-bit words, of table, including this field.
16 bits	DescriptorID	DescriptorID = 0003h identifies this as an obsolete DDM table
8 bits	EntrySize	Number of 32-bit words in each entry
8 bits	NumberEntries	Number of entries
8 bits	TableVersion	This version = 00h
8 bits	reserved1	reserved
n x m	*data* <as follows>	Table entries (size = EntrySize x NumberEntries x 32 bits). Each entry consists of the fields defined in Table 5-15.

Table 5-15 Obsolete Table Entry

Size	Name	Description
16	OrganizationID	Identifies the vendor producing the module (see Table 5-5)
16	ModuleID	Identifies the module (see Table 5-5)

Table 5-16 Structure of TCL Script Table

Size	Name	Description
16 bits	length	Length, in 32-bit words, of table, including this field
16 bits	DescriptorID	DescriptorID = 0004h identifies this as a TCL Script table
16 bits	reserved2	reserved
8 bits	ScriptVersion	This version = 00h
8 bits	reserved1	reserved
	data	**TCL Script List** – table of TCL Script for Configuration Dialogue. See section 5.3.3.2, *Module Script Table*.

5.3.2.2 Module Parameter Block

Each installed DDM has an associated module parameter block to hold user-configurable parameters.

Table 5-17 Structure of Module Parameter Block

Size	Name	Description
32 bits	MPBsize	The size of the module parameter block, in 32-bit words, including this field
16 bits	OrganizationID	The I$_2$O organization ID assigned to the vendor producing the DDM
16 bits	ModuleID	This value must match the ModuleID in the module header
32 bits	MPBversion	Used by the vendor to synchronize MPB structures when upgrading existing DDMs. When the system creates the module parameter block, it sets this value to 0000-0000h and sets MPBsize to 4. The DDM can set this to any value.
32 bits	reserved4	reserved
x32 bits	MPB data	The remainder of the module parameter block is specific to the vendor. The DDM stores any information that needs to be persistent from one power cycle to the next. Should include any information such as device tags to correlate TIDs assigned to ports and physical devices, as well as configuration information for making other connections. Each DDM vendor has its own private requirements, based on the classes of service it provides.

x32 bits means any number of 32-bit words.

5.3.2.3 System Configuration Information

Each I$_2$O device provides a description of itself that is used to create the IOP's logical configuration table. This information is specified in *Logical Configuration Table Entries* in Chapter 3.

5.3.3 Configuration Dialogue

5.3.3.1 TCL Scripts

To carry out an interaction with a human user using the HTML-based dialogue facility, a DDM must format HTML text and parse input *form* data. While this can be done in the C language, it is far easier to use text macro and scripting facilities. The scripting facility for I $_2$O allows the following:

- string substitution for internationalization of text.
- access to fields in parameter groups, to get values for display and setting values as the result of form submissions.
- formatting values obtained from those fields (e.g. display as decimal or hex integer).
- conditional text substitution based on values of fields (e.g. adds HTML command *CHECKED* if field equals a particular value).
- evaluating arithmetic and logical expressions.
- variable length loops, depending on values of fields (e.g. display a table with a variable number of rows).
- defining additional macros or procedures for higher levels of functionality.

- a dense definition of the HTML dialogue.

The I_2O scripting facility is a small subset of Tool Command Language (TCL), a very flexible and expressive language. However, it is extremely simple in its underlying syntax and, therefore, the footprint of the basic interpreter is quite small.

5.3.3.1.1 Standard TCL command support

IRTOS provides the following subset of standard TCL commands for I_2O dialogue writers.

Table 5-18: Standard TCL Commands Supported by IRTOS

Command	Description
SET	Read and write variables
UNSET	Delete variables
EXPR	Evaluate an expression
EVAL	Evaluate a TCL script
FOR	Iterate over sequential values
FOREACH	Iterate over all elements in a list
WHILE	Execute script repeatedly as long as a condition is met
BREAK	Abort looping command
CONTINUE	Skip to the next iteration of a loop
IF	Execute scripts conditionally
SWITCH	Evaluate one of several scripts, depending on a given value
FORMAT	Format a string in the style of sprintf
SCAN	Parse string using conversion specifiers in the style of sscanf
CONCAT	Join lists together
JOIN	Create a string by joining together list elements
SPLIT	Split a string into a proper Tcl list
PROC	Create a Tcl procedure
RETURN	Return from a procedure

5.3.3.1.2 Custom TCL Commands for IRTOS

In addition to the standard TCL commands previously listed, this specification defines several commands adapted from the TCL command set. The definitions of these modified TCL commands follow:

Table 5-19: Customized TCL Commands Support by IRTOS

puts *string*

puts is a standard TCL command normally used to write to the standard output stream. For I_2O, this command is modified to write into the reply buffer indicated by an ***UtilDialogGet*** message.

Source ***script***

source is a standard TCL command that reads and executes another script before executing the current script. This can include a set of common definitions and TCL procedures into multiple scripts, or for internationalization, provide alternate strings for dialogue pages for different languages. A script for each supported language can set a collection of variables to the appropriate dialogue strings in that language. To display dialogues in the appropriate language, a TCL script first sources the script containing the strings of the selected language.

Source normally reads scripts from an underlying file system. For I$_2$O, the source command is modified to read scripts from a DDM's script table.

KeysGet ***group keytype***

keysGet returns the keys of the specified group. The ***keytype*** parameter can take the values **b** or **s**: **b** indicates binary keys, and **s** indicates ASCII strings. Binary keys are converted into ASCII numeric format ("0x…") before being returned.

fieldGet ***group field [key keytype] type*** **&**

fieldSet ***group field [key keytype] type value***

fieldGet/fieldSet will get/set the value specified by the (***group***, ***field***, ***key***) triple. If the group is scalar, the ***key*** and ***keytype*** arguments are omitted. The ***type*** and ***keytype*** arguments can be **s** or **b**: **s** indicates an ASCII string type, and **b** indicates a binary type.

add ***group key keytype***

delete ***group key keytype***

add/**delete** adds or deletes a row with the specified ***key*** to/from a table group. The ***keytype*** argument can be **s** or **b**, indicating whether the key type is string or binary. Once a row is added to a table, **fieldSet** calls can set the values of fields in that row.

clear ***group***

clear removes all rows from the specified table group.

The IRTOS provides pre-defined TCL commands to simplify producing standard HTML items such as menus, check boxes, radio buttons, tables, and so on, and for standardized parsing of form data. A device may make use of these commands simply by sourcing the built-in TCL script defining these commands.

The actual content of this pre-defined TCL script is beyond the scope of this specification.

5.3.3.2 Module Script Table

The DDM module descriptor header includes dialogue scripts. The format of the TCL script table is shown below:

31 3 24	23 2 16	15 1 8	7 0 0	offset
0004h		Length		0
reserved	ScriptVersion	reserved		4
Offset from start of table for Set0 (=a)				8
Offset from start of table for Set1 (=b)				12
:				:
Offset from start of table for Setn (=c)				:
Offset from start of table for Set0 Page0 (=d)				a
Offset from start of table for Set0 Page1 (=e)				a+4
:				:
Offset from start of table for last page of Set0 (=f)				:
Offset from start of table for Set1 Page0 (=g)				b
Offset from start of table for Set1 Page1				b+4
:				:
Offset from start of table for last page of Set1				:
Offset from start of table for Setn Page0				c
Offset from start of table for Setn Page1				c+4
:				:
Offset from start of table for last page of Setn				:
TCL Script for Set0 : Page0				d
:				:
TCL Script for Set0 Page1				e
:				:
TCL Script for Last Page of Set0				f
:				:
TCL Script for Set1 Page0				g
:				:

Figure 5-2. TCL script table format

Two numbers identify each script: a *set* number and a *page* number. The set number distinguishes different sets of pages, since a DDM may need to provide several sets of dialogues if it creates devices of different classes. The location of sets and pages within the table is determined by 32-bit pointers relative to the start of the table. The scripts themselves are byte arrays and are byte-aligned within the table.

The scripts are in the module's header stored in the IOP's non-volatile store. Thus, dialogue scripts never occupy RAM. The IRTOS locates scripts by its set number and page number. DDM writers simply identify dialogue scripts by its set and page number.

5.4 IRTOS: I$_2$O Real-Time OS

This section specifies IRTOS, a special-purpose real-time OS designed specifically to support high-speed, low-overhead I/O operations.

5.4.1 Purpose

I$_2$O provides a framework for off-loading I/O processing from the central CPU(s) to one or more dedicated I/O processors. Thus, the I$_2$O architecture splits I/O drivers into two components: OSMs that run on the CPU(s) under the host OS' operating system, and DDMs that run on the IOP under a dedicated operating system called IRTOS.

IRTOS is specified by an API that defines the services available to DDMs and that the DDM is expected to supply. IRTOS does not refer to any specific implementation of the specification; many different implementations can qualify as IRTOS-compliant if they provide the exact API specified here.

Use of IRTOS is not required by integrated intelligent devices that provide their own I/O processor. In that case, the device is I$_2$O compliant if it fully adheres to the I$_2$O shell communication protocols and those for each specific I/O class.

However, to be I$_2$O compliant, system vendors that provide open I/O processors for execution of arbitrary device drivers must provide the IRTOS environment. Device vendors that provide the I$_2$O compliant drivers to run on those I/O processors must create DDMs that conform to the IRTOS environment. This level of IRTOS compliance allows the compatibility and interoperability of drivers and system platforms that I$_2$O intends to provide.

5.4.2 IRTOS Overview

IRTOS is organized around two primary concepts: objects and events. Objects offer a uniform way to provide system services. They also facilitate tracking and reclaiming system resources. Objects provide a mechanism for configuration-time or run-time binding of variants of system services to specific users, without requiring their recompilation or reinstallation.

The IRTOS driver abstraction is based on an event-driven model of device drivers. In this model, all inputs to the driver are in the form of events that are queued to the driver. The driver writer defines an event handler for each event that the driver can receive. The IRTOS package handles the bulk of the bookkeeping for the driver. IRTOS translates I$_2$O request messages, device interrupts, DMA completion, and timer expirations into events, and invokes the appropriate DDM handler functions.

The following figure illustrates the primary IRTOS objects, event facilities, and how they are related. Details of each component are in the sections below.

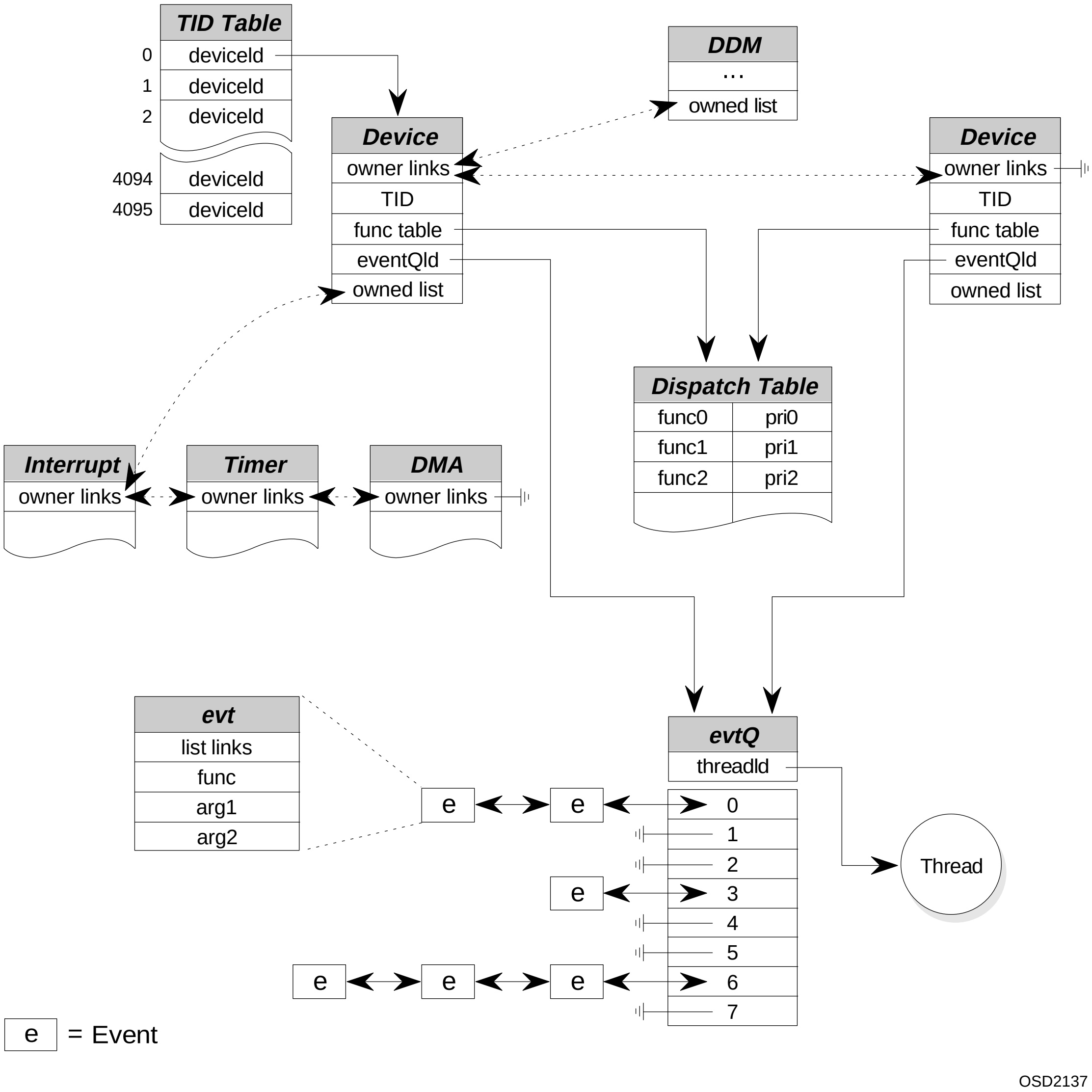


Figure 5-20 IRTOS Component Overview

5.4.2.1 IRTOS API Conventions

This section discusses the API naming conventions and error handling.

5.4.2.1.1 Naming Conventions

All IRTOS API functions follow a uniform naming convention. IRTOS functions are named as follows:

i2o<noun><verb>

Because IRTOS functionality is organized around various types of system objects, the **<noun>** is usually the name of the object class and the **<verb>** is a function or property that applies to that class.

IRTOS data types are given C `typedefs` and are written in uppercase with the prefix `I2O:`

```
I2O_STATUS
```

IRTOS constants and enumeration values are likewise written in uppercase with the prefix *I2O* and an additional prefix identifying the type to which they belong:

I2O_STS_INVALID_OBJECT_ID

5.4.2.1.2 Error Handling

IRTOS provides an error handling mechanism that is simple, uniform across all API functions, and significantly increases the robustness of IRTOS-based IOPs. All IRTOS API functions (except those few that cannot possibly incur errors) take as their last argument a pointer to a status variable of type I2O_STATUS. If any error is encountered in performing an IRTOS function, the IRTOS error code is returned in this status variable. If an IRTOS function completes, the status variable is *not* modified. Thus, the user should initialize the status variable to *I2O_STS_OK* before calling the IRTOS function.

However, if a NULL pointer is specified for the pointer to the status variable, then IRTOS automatically takes a predetermined error action if an error is encountered. Possible error actions include ignoring the error and continuing, suspending execution of the calling thread and invoking a debugger, or calling a user-specified error handler. The default error action can be specified for a thread by calling the **i2oThreadErrorActionSet()** function (see section 5.4.16 *Threads*).

The symbol *I2O_NO_STATUS* is defined as a NULL pointer to a variable of type I2O_STATUS. Thus, the following example invokes the default action if any error is encountered when calling **i2oObjContextGet()**:

```
context = i2oObjContextGet (objId, I2O_NO_STATUS);
```

Most errors encountered in IRTOS API functions represent a program error or a corruption of system or DDM data structures. For example, an invalid object ID specified to any of the object functions (such as the call to **i2oObjContextGet()** shown above) usually results from a bug in the DDM or a DDM data structure corrupted by a bug in another program. A DDM encountering such a fault should immediately stop execution or invoke comprehensive error recovery, such as reloading the DDM. However, checking the status of every IRTOS API call in the DDM code adds considerable overhead and obfuscates the DDM code. IRTOS automatic error handling provides a simple alternative that can stop errant code as soon as it is detected.

By specifying *I2O_NO_STATUS* as the status pointer, the caller effectively tells the IRTOS “I’m not going to be checking the status of this function, so please prevent me from continuing on if an error is encountered in this call.”

Table 5-20 IRTOS Error Actions

Value	Description
I2O_ERR_ACT_DEFAULT	Take thread or system-specified default error action
I2O_ERR_ACT_IGNORE	Ignore error and continue
I2O_ERR_ACT_USER	Call user-specified user function
I2O_ERR_ACT_DEBUG	Suspend program and invoke debugger

Table 5-21 IRTOS Status Pointer

Value	Description
I2O_NO_STATUS	When specified as value of status return pointer (&status) in IRTOS API functions, indicates that thread or system-specified error action should be taken if an error is encountered
ANY OTHER VALUE	When specified as value of status return pointer (&status) in IRTOS API functions, indicates that the function must return a status code of type I2O_STATUS at that location.

Invoking IRTOS Error Handling

The IRTOS status return and error handling policy described above is directly implemented by the function **i2oErrorSet()**. This function takes as arguments an error code and a pointer to a status variable. If the pointer is not NULL, then the error code is set in the status variable. If the pointer is NULL, then the current IRTOS default error action is invoked. Thus, a DDM writer can implement a subroutine that provides IRTOS-style error handling, as in the following example:

```
void myHandler (int arg1, I2O_STATUS *pCallersStatus)
      {
      ...
      if <error is detected>
           {
           i2oErrorSet (MY_ERROR_CODE, pCallerStatus);
           return; /* may not reach here, depending on error action */
           }
      {
```

Another IRTOS function, **i2oErrorAction(),** allows invoking IRTOS error actions directly. This function takes as arguments an error code (which can be logged or reported by the error action) and the type of error handling desired. Specifying *I2O_ERR_ACT_DEFAULT* causes the current default error action to be invoked. This is often used when a DDM needs to handle error conditions from one IRTOS function, but not others. In this case, the DDM calls the IRTOS function with a pointer to a status variable. When the call returns, the DDM tests for special cases. If it finds no special cases, the DDM calls **i2oErrorAction()** for default error handling.

```
I2O_STATUS status = I2O_STS_OK;   /* initialize status to OK */
value = i2oBusRead (busId, space, addr, &status);
if (status != I2O_STS_OK)
       {
       if (status == I2O_BUS_INVALID_ADDRESS)
              … reset device …
       else
              {
              /* let IRTOS handle all other errors */
              i2oErrorAction (status, I2O_ERR_ACT_DEFAULT);
              return; /* may not reach here, depending on error action */
              }
       }
```

Also, DDM writers can call **i2oErrorAction()** to invoke specific error actions other than the current default.

Table 5-22 IRTOS Error Action Functions

Returns	API Function Call	Description
void	**i2oErrorSet** (errorCode, &status)	Set error code or invoke error action
void	**i2oErrorAction**(errorCode, errorAction)	Invoke error action

Parameter	Type	Description
errorCode	I2O_STATUS	code of error that occurred (see Table 5-23)
&status	I2O_STATUS *	variable to receive error code
errorAction	I2O_ERROR_ACTION	error action to perform (see Table 5-20)

Table 5-23 I2O_STATUS – IRTOS Status Values

Value	Description
I2O_STS_OK	No error
I2O_STS_INVALID_OBJ_ID	The object ID is not a valid object or of the required class
I2O_STS_INVALID_OWNER_ID	The specified owner ID is not a valid device ID
I2O_STS_NOT_ISR_CALLABLE	The API function was invoked from an Interrupt Service Routine (see section 5.4.12 for API functions that can be called from ISRs)
TBD	complete list will be provided

5.4.2.2 I$_2$O Data Types

The following table summarizes the data types defined by the IRTOS API.

Table 5-24 I$_2$O Data Types

I$_2$O Data Type	Definition
BOOL	Boolean: TRUE, FALSE
int	Default integer
INT8	8-bit integer
INT16	16-bit integer

I$_2$O Data Type	**Definition**
INT32	32-bit integer
INT64	64-bit integer
I$_2$O_ADAPTER_ID	ID of adapter object
I$_2$O_ADDR32	32-bit address value
I$_2$O_ARG	Arbitrary argument
I$_2$O_BBU_ATTR	Battery backup attribute (see Table 5-46)
I$_2$O_BBU_STATUS	Current condition of battery backup (see Table 5-47)
I$_2$O_BUS_ID	ID of bus object
I$_2$O_BUS_SPACE	Bus space: *I2O_BUS_SPC_MEMORY* *I2O_BUS_SPC_IO*
I$_2$O_COUNT	Positive count of items
I$_2$O_DDM_ID	ID of DDM object
I$_2$O_DDM_TAG	Concatenation of I$_2$O organizationId and moduleId
I2O_DEV_BIOS_INFO	Device BIOS correlation information
I2O_DEV_CHANGE_INDICATOR	Change indicator of device info in LCT
I2O_DEV_CLASS	Device class
I2O_DEV_EVENT_CAPABILITIES	Device event capabilities in LCT
I2O_DEV_FLAGS	Device flags in LCT
I2O_DEV_ID	ID of device object
I2O_DEV_IDENTITY_TAG	Unique device identifier in LCT
I2O_DEV_SUBCLASS	Device subclass information (class specific)
I2O_DISPATCH_TBL_ID	ID of dispatch table object
I2O_DMA_CANCEL_MODE	Mode of cancel operation (see Table 5-58)
I2O_DMA_CREATE_FLAGS	Option flags when creating DMA objects (see Table 5-54)
I2O_DMA_ID	ID of DMA object
I2O_DMA_XFER_FLAGS	Option flags when queuing DMA transfers (see **Table 5-57 DMA Transfer Flag Values**)
I2O_ERROR_ACTION	Automatic error action (see Table 5-20 IRTOS Error Actions)
I2O_ERROR_FUNC	Error action function
I2O_EVENT_HANDLER	Event handler function
I2O_EVENT_PRI	Event priority: 0 (highest) – 7 (lowest)
I2O_EVENT_PRI_MASK	Event priority mask, LSB = priority 0
I2O_EVENT_QUEUE_ID	ID of event queue object
I2O_FRAME	I$_2$O message frame
I2O_FUNC_ENTRY	Function description: {message Function code, message handler, priority}
I2O_INITIATOR_CONTEXT	Initiator context field in request and reply messages
I2O_INT_ID	ID of interrupt object
I2O_INT_LOCK_KEY	Interrupt state key
I2O_IOP_CONFIG_FLAGS	Flags indicating configuration of this IOP
I2O_IOP_CONFIG_INFO	Structure containing configuration constants for this IOP

I$_2$O Data Type	Definition
`I2O_ISR_HANDLER`	Interrupt service routine handler
`I2O_LCT_INFO`	Logical configuration table entry descriptor
`I2O_MEM_ACCESS_ATTR`	Access attributes of memory (see Table 5-41)
`I2O_MEM_CACHE_ATTR`	Cache attributes of memory (see Table 5-42)
`I2O_MEM_PART_ID`	ID of memory partition object
`I2O_MODULE_PARAM_BLOCK`	Module parameter block
`I2O_OBJ_CONTEXT`	Arbitrary context value for object
`I2O_OBJ_ID`	ID of object (unspecified type)
`I2O_OBJ_NAME`	Name string of object (ASCII string of printable non-whitespace characters)
`I2O_OWNER_ID`	ID of owner device object (equated to `I2O_DEV_ID`)
`I2O_PHYS_LOCATION`	Adapter physical location descriptor
`I2O_PIPE_ID`	ID of pipe object
`I2O_PIPE_OPTIONS`	Pipe options (see Table 5-65)
`I2O_PIPE_PRI`	Priority of message on pipe: *I2O_PIPE_PRI_NORMAL* *I2O_PIPE_PRI_URGENT*
`I2O_SEM_B_STATE`	Initial state of semaphore object: *I2O_SEM_FULL* *I2O_SEM_EMPTY*
`I2O_SEM_ID`	ID of semaphore object
`I2O_SEM_OPTIONS`	Semaphore options (see Table 5-63)
`I2O_SG_ELEMENT`	Element of scatter-gather list
`I2O_SIZE`	Value representing size (length) of item in bytes
`I2O_STATIC_MSG_ID`	ID of static message object
`I2O_STATUS`	Status value (see Table 5-21)
`I2O_THREAD_FUNC`	Initial thread function
`I2O_THREAD_ID`	ID of thread object (NULL means *self* when passed as parameter)
`I2O_THREAD_OPTIONS`	Thread options (see Table 5-60)
`I2O_THREAD_PRI`	Thread priority: 0 (highest) – 255 (lowest)
`I2O_TID`	I$_2$O target identifier: 0 - 4095
`I2O_TIMER_ID`	ID of timer object
`I2O_TRANSACTION_CONTEXT`	Transaction context field in request and reply messages
`I2O_USECS`	Number of microseconds

5.4.2.3 Objects

All IOP facilities in IRTOS are encapsulated as objects. Every object has:

- an ID the user and the system use to refer to the object
- virtual functions for create, destroy, and other generic object functions
- an owner device to which the object belongs
- a name (optional)
- a user context value (optional)

IRTOS provides objects for the following facilities:

- hardware access:
  - interrupt objects for connecting, acknowledging, vectoring, enabling, and disabling
  - DMA objects for allocating, initiating, and terminating DMA channels
  - bus objects for accessing hardware and system memory
  - adapter objects for initializing and mapping physical devices
- I_2O message protocol package:
  - initializing and allocating channels
  - dispatching request messages to identified targets
  - reply messages
  - low-level formatting and parsing
  - routing and forwarding facilities
- multi-threading package:
  - thread manipulation for creation, deletion, setting priorities, etc.
  - memory management
  - mutexes and semaphores
  - inter-thread communication pipes
  - event queues
  - virtual timers for setting watchdogs, delays, and timeouts

The following table lists the classes of objects defined in IRTOS.

Table 5-25 Classes of Objects Defined in IRTOS

Class	Description
DDMs	DDM description and manipulation
devices	Device description and manipulation
dispatch tables	Tables of message functions for I_2O request message handling
DMA	DMA allocation and execution (i.e., data movement)
hardware adapters	Get information about specific hardware devices
interrupts	Interrupt handling
memory sets	Allocating and deallocating memory from memory sets
page sets	Allocating and deallocating pages from page sets
physical buses	Read and write to each bus and translate addresses between buses
pipes	Send and receive variable length data between threads
semaphores	Binary, counting, and mutex semaphores or synchronization and mutual exclusion
static messages	High-speed communication of pre-determined, unchanging messages
threads	Independent, prioritized, preemptable threads of program execution
timers	Timeouts, timestamps, one-shot and periodic timers

The table below lists the generic object functions. These functions can be invoked on any of the IRTOS objects.

Table 5-26 IRTOS Object Functions

Returns	API Function Call	Description
void	**i2oObjDestroy** (objId, &status)	Destroy object
ownerId	**i2oObjOwnerGet** (objId, &status)	Get owner of object
void	**i2oObjOwnerSet** (objId, ownerId, &status)	Set owner of object **{** XE "**i2oObjOwnerSet()**" **}**
pName	**i2oObjNameGet** (objId, &status)	Get name of object
void	**i2oObjNameSet** (objId, pName, &status)	Set name of object
context	**i2oObjContextGet** (objId, &status)	Get user context from object
void	**i2oObjContextSet** (objId, context, &status)	Set user context in object

Parameter	Type	Description
&status	I2O_STATUS *	Variable to receive error code
context	I2O_OBJ_CONTEXT	User context value (see 5.4.2.3.3), typically a pointer to a data structure
objId	I2O_OBJ_ID	ID of object
ownerId	I2O_OWNER_ID	ID of owner device object (see 5.4.2.3.1)
pName	I2O_OBJ_NAME *	Pointer to name string assigned by user

5.4.2.3.1 Object Ownership

Each IRTOS object is *owned* by an I$_2$O device. The owner device is specified as a parameter in each IRTOS object *create* function. The primary use for this is in resource tracking and reclamation. When a device is deleted, all objects it owns are automatically deleted as well. An IRTOS may also use ownership to limit resource utilization by device. Object ownership by devices is discussed in section 5.4.3.7.

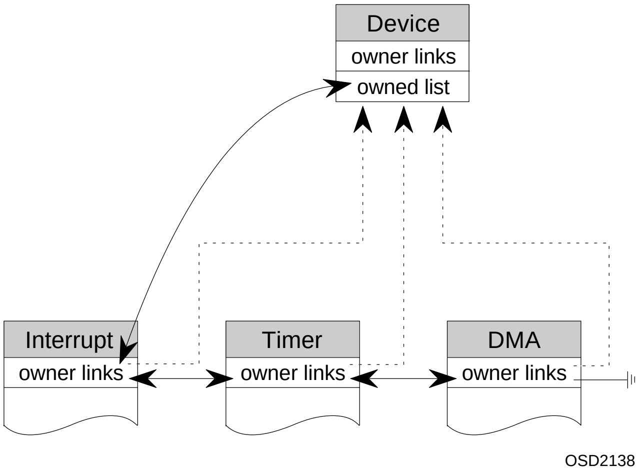


Figure 5-21 Example of IRTOS Object Ownership

5.4.2.3.2 Object Names

Any IRTOS object can be assigned a user-readable name. This is primarily used to identify the object to the user or DDM developer. The IRTOS uses object names, if set, in reporting error messages, configuration dialogues, and so on. An object name must be an ASCII string of printable, non-whitespace characters.

An object name can be set or read via the **i2oObjNameSet()** and **i2oObjNameGet()** functions. The name string passed as a parameter to the **i2oObjNameSet()** function is copied by the system and so can be altered or destroyed after the function returns.

5.4.2.3.3 User Context of IRTOS Objects

DDM code often needs to relate an IRTOS object to another DDM object or data structure. To facilitate this, every IRTOS object has a four-byte context value that DDMs can freely use to hold any arbitrary value. Often this value is a pointer to an internal DDM data structure.

The user context of any object can be set or read via the **i2oObjContextSet()** and **i2oObjContextGet()** functions. Also, as noted in section 5.4.2.5, IRTOS objects that post events (such as a DMA object) supply that object's user context as an argument when posting an event (such as DMA completion). The DDM specifies the user context when it creates any of these objects.

5.4.2.4 Events

The IRTOS driver model is called event-driven because all inputs relevant to a device are encapsulated in a common event structure and delivered to the appropriate driver via a prioritized event queue. Drivers consume the incoming events by calling the appropriate event handler function for each. All event handlers must be short, non-blocking functions so that drivers do not wait, except when waiting for events.

An event is characterized by the following elements:

- the function to call that handles the event (i.e., the specific event handler)
- arguments to that function
- its event queue
- its priority in the event queue

An event queue is prioritized. Associated with each event queue is a single thread that dequeues the events and executes the appropriate event handlers, one at a time. Thus, handling events on a given event queue is serialized by the single thread servicing it. This eliminates the need for mutual exclusion mechanisms on device data structures within event handler functions, since no two event handlers can execute simultaneously for a given device.

There are eight event priority levels. The thread associated with the event queue always dequeues the highest-priority event queued. If no events are currently queued (at an enabled priority level), the thread is blocked until either such an event posts to that queue, or a priority level is enabled for which events are queued.

The **i2oEventQPriEnableSet()** function individually enables and disables event priority levels and the **i2oEventQPriMaskSet()** function operates on all priority levels. If a priority level is disabled, then no events at that level are dequeued by the event queue's thread until the priority level is enabled. Thus, events of a particular priority can be held off in the event queue. The **i2oEventQPriEnableGet()** and **i2oEventQPriMaskGet()** functions identify which priority levels are enabled. In the priority bit masks, the least significant bit corresponds to priority 0 (highest), 1 indicates an enable priority, and 0 indicates a disabled priority.

The **i2oEventQPriPending()**function returns a bit mask of priorities on which events are pending. In this mask, the least significant bit corresponds to priority 0 (highest), 1 indicates a priority with events pending, and 0 indicates a priority with no events pending.

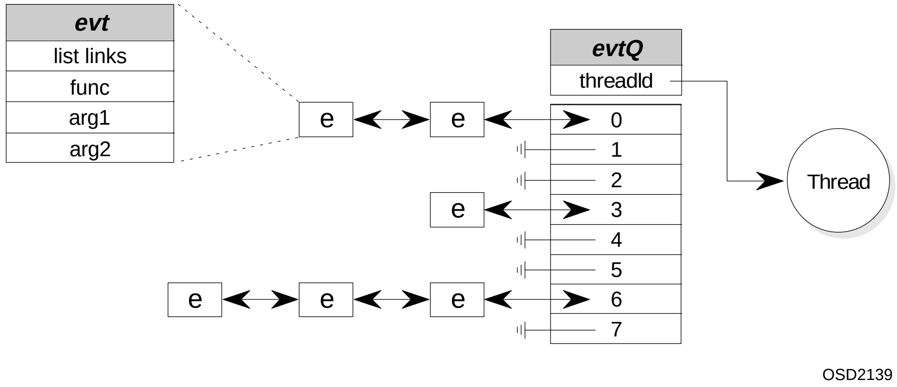


Figure 5-22 IRTOS Events and Event Queues

Table 5-27 Event Queue Functions

Returns	API Function Call	Description
`evtQId`	**i2oEventQCreate** `(ownerId, threadOptions, threadStackSize, &status)`	Create event queue
`threadId`	**i2oEventQThreadGet**`(evtQId, &status)`	Get ID of event queue thread
`enableFlag`	**i2oEventQPriEnableGet** `(evtQId, evtPri, &status)`	Get priority enable state
`void`	**i2oEventQPriEnableSet** `(evtQId, evtPri, enableFlag, &status)`	Set priority enable state
`evtPriMask`	**i2oEventQPriMaskGet** `(evtQId, &status)`	Get priority enable mask
`void`	**i2oEventQPriMaskSet** `(evtQId, evtPriMask, &status)`	Set priority enable mask
`evtPriMask`	**i2oEventQPriPending** `(evtQId, &status)`	Get mask of priorities with queued events
`void`	**i2oEventQPost** `(evtQId, evtPri, evtHandler, evtArg1, evtArg2, &status)`	Post event to queue

Parameter	Type	Description
`&status`	`I2O_STATUS *`	Variable to receive error code
`enableFlag`	`BOOL`	TRUE = priority level enabled, FALSE = priority level disabled
`evtArg1`	`I2O_ARG`	First argument to event handler
`evtArg2`	`I2O_ARG`	Second argument to event handler
`evtHandler`	`I2O_EVENT_HANDLER`	Pointer to event handler called when event is dispatche*d
`evtPri`	`I2O_EVENT_PRI`	Event priority level, 0 (highest) - 7 (lowest)
`evtPriMask`	`I2O_EVENT_PRI_MASK`	Bit mask of priority levels, LSB = priority 0, 1 = enabled/pending, 0 = disabled/none pending

Parameter	Type	Description
evtQId	I2O_EVENT_QUEUE_ID	ID of event queue object
ownerId	I2O_OWNER_ID	ID of owner device object (see 5.4.2.3.1)
threadId	I2O_THREAD_ID	ID of thread that services event queue
threadOptions	I2O_THREAD_OPTIONS	Event queue thread options -: no options defined at this time
threadStackSize	I2O_SIZE	Size of event queue thread's stack in bytes

Certain IRTOS functions may delay the request. Instead of blocking the caller until the request completes, these functions post events when they finish. These functions generally take additional parameters such as the event priority, event handler, and arguments to that handler. Such functions return as soon as all immediate processing completes and the waiting condition has been encountered. In this case, the operation is queued in the system. When the waiting condition is removed and the requested operation is complete, the event specified in the original call is posted. The table below lists the IRTOS functions that may require waiting, and therefore post events.

Table 5-28 IRTOS Functions That Post Events

API Function Call	Description
i2oDmaXfer()	Do DMA transfer
i2oDmaXferFrag()	Do DMA transfer of list fragment
i2oDmaXferList()	Do DMA transfer of list
i2oIntEventPost()	Post event from interrupt service routine
i2oPageAllocContig	Allocate initial contiguous block of memory
i2oPageBbuNotify	Notify on crossing of battery backup threshold
i2oStaticMsgCreate()	Create static message frame
i2oTimerRepeat()	Start periodic timer
i2oTimerStart()	Start one-shot timer

For event handlers to execute promptly, they must be short and not block indeterminately. Thus, all waiting in an event-driven program such as a DDM is for an event, rather than at a system call coded into an event handler. For example, suppose a driver executing an event handler requires a free buffer, but none is available. If the driver is blocked in the event handler waiting for a free buffer, then the driver does not respond to incoming events until the buffer becomes available. This could produce poor response times, at best, and complete deadlocks, at worst.

If an event handler blocks on a mutex semaphore of a known, deterministic set of critical regions is acceptable, because this does not constitute indefinite waiting. However, the mutex must be protected from priority inversion by the priority inheritance protocol (see section 5.4.18, *Semaphores*).

Using the IRTOS model, the DDM operates as follows:

- In general, as little device processing as possible takes place at interrupt level, for example, in interrupt service routines (ISR). Except where microsecond-level response to devices is required, ISRs simply dispatch events to the appropriate event queues, which are then serviced by the appropriate driver threads.

- As implemented in the IRTOS framework, a driver thread follows a simple event handling model:

```
loop:
    {
    wait for next event
    dispatch to appropriate function to handle event
    }
```

- In general, handling each event by driver threads must be short and not block waiting for specific interactions. Drivers should do all waiting in the main event-handling loop. This ensures that the drivers can always examine incoming request messages, even during I/O operations.
- The drivers can use various system objects such as timers and DMA objects to initiate operations and receive event notification on completion.
- Drivers can use other IOP primitives to implement more sophisticated device control. For example, a driver could spawn additional threads for asynchronous processing of algorithms that do not fit in the discrete event handling model of the basic drivers (e.g., elevator sorting, caching, and network protocols).
- Drivers invoke other drivers by sending I$_2$O requests and receiving the corresponding replies as events. Thus, hierarchies of drivers can be constructed (e.g., a RAID driver that utilizes raw disk drivers).

The diagram below shows examples of events queued to two hierarchically-related devices.

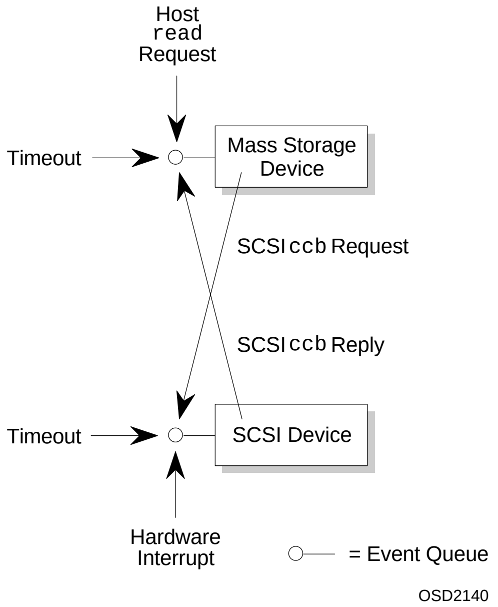


Figure 5-23 Hierarchical Event Queues

5.4.2.5 Event Handlers

All I$_2$O event handlers have the same basic declaration:

```
void evtHandler (arg1, arg2)
```

The parameters arg1 and arg2 are 32-bit values whose content depends on the event handler. When an event reaches the top of the event queue, the event dispatcher retrieves the address of the event handler and its arguments from the event structure. It calls that event handler with its arguments.

5.4.2.5.1 Specific IRTOS Object Event Handlers

Several IRTOS objects provide asynchronous notification to drivers in the form of events. The event parameters include:

- the event queue to post the event to
- the event priority
- the pointer to the event handler
- perhaps arguments to the handler, specified either when the object is created or when a particular function requiring event posting is invoked on that object.

For example, for a timer object, the event queue ID is specified when the timer object is created, and the event priority, handler, and an argument are specified when the timer is started.

Objects that post events require specific event handler functions that take specific arguments defined by the object. By convention, these event handlers all receive the object's user context as the first argument; the object's user context is initialized by an argument to the object's creation function. The IRTOS considers the context value an arbitrary four-byte value and never touches it except to pass it as an argument to the object event handlers. Typically, the context value is a pointer to a private data structure within the user's program.

The following table summarizes the IRTOS objects that post events and the declaration of their event handler functions.

Table 5-29 Event Handler Functions for IRTOS Objects

Object	Initiating Action	Event Handler
Battery Backup	**i2oPageBbuNotify()**	`void bbuEvtHandler (pageContext, bbuEvtArg)`
Device	receiving request or reply msg	`void msgHandler (devContext, pFrame)`
Dma	**i2oDmaXfer()**, **i2oDmaXferList() and** **i2oDmaXferFrag()**	`void dmaEvtHandler (dmaContext, dmaStatus)`
Interrupt	**i2oIntEventPost()**	`void intEvtHandler (intContext, intEvtArg)`
Page Set	**i2oPageAllocContig()**	`void allocEvtHandler (pageContext, allocEvtArg)`
Timer	**i2oTimerStart()** and **i2oTimerRepeat()**	`void timerEvtHandler (timerContext, timerEvtArg)`
Static Msg	**i2oStaticMsgCreate()**	`void smEvtHandler (smContext, smEvtArg)`

5.4.3 DDMs and Devices

IRTOS provides two objects that represent the basic entities each driver provides: the DDM itself and the devices it services. An IRTOS DDM object represents all components associated with a running driver: the actual DDM code and global data, and all I_2O devices and OS objects created by the DDM, such as threads, timers, and semaphores.

An IRTOS device object represents the I_2O addressable entities created by a DDM. Each device is assigned an I_2O target ID, i.e., a TID. One DDM generally creates several devices, possibly of different classes. Every DDM has one device that represents the DDM itself.

5.4.3.1 DDM Initialization Function

Every DDM has a single initialization function identified in the DDM object module. The IRTOS calls this function whenever it loads the driver. The function receives a single argument; a pointer to the DDM's Module Parameter Block (MPB). See the discussion of **i2oDdmMpbStore()** in section 5.4.3.2.

DDM initialization function declaration:

```
     void         ddmInit (pMpb)
```

5.4.3.2 DDM Management Functions

Every DDM must create a single object by calling **i2oDdmCreate()** in the DDM's initialization function. This function creates the DDM object and several associated objects, and causes the IRTOS to add an entry to its logical configuration table identifying the DDM. It uses the values provided in the lctInfo parameter (see section 5.4.3.4).

In addition to the DDM object, this function also creates the following:

- an event queue, for events posted to the DDM
- a thread to service the event queue, using the threadOptions and threadStackSize parameters (see section 5.4.16)
- a *dispatch table* is created, using the pFuncArray and numFuncs parameters, that holds the message handler entry points for each message to which the DDM responds (see section 5.4.6)
- a DDM class *device* object (described below) that represents the DDM itself. The IRTOS and host OSMs use this device to communicate with the DDM itself, as opposed to its I/O devices. The IRTOS uses the ddmTag parameter to identify the DDM. Then it assigns the same TID to the DDM device on each reboot of the IOP (see section 5.4.3.5). The devContext argument sets the context of the DDM device, and thus, becomes the first argument to the message handlers. The ID of the device object for the DDM is obtained by calling **i2oDdmDevGet()**.

The IRTOS also provides each DDM with a Module Parameter Block (MPB), a single, expandable block of non-volatile storage. The first time the driver is loaded, the MPB is empty. Subsequently, the DDM can store arbitrary data in its MPB using **i2oDdmMpbStore()**. This function takes a pointer to a scatter-gather list of data that is to replace the old MPB. That data then goes in the MPB provided to the DDM's initialization function (see section 5.4.3.1).

Table 5-30 DDM Management Functions

Returns	API Function Call	Description
ddmId	**i2oDdmCreate** (ddmTag, i2oVersion, &lctInfo, devContext, threadOptions, threadStackSize, pFuncArray, numFuncs, &status)	Create DDM object
devId	**i2oDdmDevGet** (ddmId, &status)	Get device ID of DDM
void	**i2oDdmTidRelease** (ddmId, tid, &status)	Release TID assigned to DDM
void	**i2oDdmMpbStore** (ddmId, pMpbList, &status)	Store module parameter block

Parameter	Type	Description
&lctInfo	I2O_LCT_INFO *	Pointer to LCT Info structure for DDM device (see Table 5-32)
&status	I2O_STATUS *	Variable to receive error code
ddmId	I2O_DDM_ID	ID of DDM object
ddmTag	I2O_DDM_TAG	Concatenation of the driver's I_2O OrganizationID and the ModuleID (see Table 5-5).
devContext	I2O_OBJ_CONTEXT	User context value (see 5.4.2.3.3), typically a pointer to a data structure
devId	I2O_DEV_ID	ID of DDM's device object; created by **i2oDdmCreate()** function
i2oVersion	int	Version of the I_2O specification under which the device operates. (0h for this version).
numFuncs	I2O_COUNT	Number of entries in pFuncArray

Parameter	Type	Description
pFuncArray	I2O_FUNC_ENTRY *	Pointer to array of dispatch function entries: {funcCode, message handler, priority} (see Table 5-33)
pMpbList	I2O_SG_ELEMENT	Pointer to a scatter-gather list for the module parameter block for storage (see Table 5-17).
threadOptions	I2O_THREAD_OPTIONS	DDM device event queue thread options - reserved - no options defined at this time.
threadStackSize	I2O_SIZE	Size of stack in bytes for the thread created for the event queue.
tid	I2O_TID	TID previously assigned to DDM

5.4.3.3 Device Management Functions

After creating the DDM object, a driver creates device objects for each I/O device it will control. Usually these are the response to *ADAPTER_ATTACH* or *DEVICE_ATTACH* messages.

To create a device, the DDM calls the **i2oDevCreate()** function, specifying the ID of the event queue to post events containing messages for that device, the dispatch table for message handler functions (see section 5.4.6), and the LCT information for the device (see section 5.4.3.4). The event queue can be created either automatically, as part of **i2oDdmCreate()**, or separately.

The **i2oDevUserTidSet()** function allows the DDM to set the UserTid field in the LCT and the associated claim flags. This is the response to a *CLAIM* message.

The **i2oDevLctFlagsSet()** function allows the DDM to set the Flags field of the LCT. This is used to set the DialogRequest flag, for example. Only those bits that are *1* in the deviceFlagsMask parameter are set to the value of the corresponding bits in the deviceFlags parameter.

Other device management functions allow the DDM information about a device object, including:

- its assigned TID
- the ID of its event queue
- its dispatch table
- its LCT entry.

Table 5-31 Device Management Functions

Returns	API Function Call	Description
devId	**i2oDevCreate** (ownerId, devContext, evtQId, dispatchTblId, &lctInfo, &status)	Create device object
tid	**i2oDevTidGet** (devId, &status)	Get TID of device
evtQId	**i2oDevEventQGet** (devId, &status)	Get event queue for device
dispatchTblId	**i2oDevDispatchTblGet** (devId, &status)	Get dispatch table for device
void	**i2oDevLctInfoGet** (devId, &lctInfo, &status)	Get LCT entry for device
void	**i2oDevLctFlagsSet** (devId, deviceFlags, deviceFlagsMask, &status)	Set deviceFlags in LCT
void	**i2oDevUserTidSet** (devId, userTid, claimFlags, &status)	Set UserTID and claimFlags in LCT

Parameter	Type	Description
&lctInfo	`I2O_LCT_INFO *`	Pointer to LCT_Info structure for this device (see Table 5-32)
&status	`I2O_STATUS *`	Variable to receive error code
claimFlags	`I2O_DEV_FLAGS`	Claim bits in device flags, to be set in LCT
devContext	`I2O_OBJ_CONTEXT`	User context value (see 5.4.2.3.3), typically a pointer to a data structure
deviceFlags	`I2O_DEV_FLAGS`	Device flags, to be set in LCT
deviceFlagsMask	`I2O_DEV_FLAGS`	Mask of bits to be altered in device flags in LCT
devId	`I2O_DEV_ID`	ID of device object
dispatchTblId	`I2O_DISPATCH_TBL_ID`	ID of dispatch table for this device
evtQId	`I2O_EVENT_QUEUE_ID`	ID of event queue for this device
ownerId	`I2O_OWNER_ID`	ID of owner device object, typically the DDM device
pageSize	`I2O_SIZE`	Hardware page size of system on which TID resides
tid	`I2O_TID`	TID to be assigned to this device, 0 if IRTOS is to make initial assignment (see 5.4.3.4)
userTid	`I2O_TID`	TID of user of this device, to be set in LCT

5.4.3.4 LCT Information

The IRTOS creates an entry in the logical configuration table (LCT) for each device, including the DDM class device implicitly created by **i2oDdmCreate()**.Table 5-32 details the origin of the values of each field in the LCT. Most information comes from the lctInfo structure parameter supplied to the **i2oDdmCreate()** and **i2oDevCreate()**functions. Specifically, the fields listed as *Set by*, ***i2oDevCreate()*** / ***i2oDdmCreate()*** are copied to the LCT by those calls. The other fields are ignored in those calls. Formal specification of these fields is in Chapter 3.

Table 5-32 LCT Info Structure

Member	Type	Set by
LocalTID	I2O_TID	IRTOS or i2oDevCreate() (see 5.4.3.4)
Flags	I2O_DEV_FLAGS	i2oDevLctFlagsSet() and i2oDevUserTidSet()
Class	I2O_DEV_CLASS	i2oDevCreate() / i2oDdmCreate()
Subclass	I2O_DEV_SUBCLASS	i2oDevCreate() / i2oDdmCreate()
UserTID	I2O_TID	i2oDevUserTidSet()
ParentTID	I2O_TID	i2oDevCreate() / i2oDdmCreate()
IdentityTag	I2O_DEV_IDENTITY_TAG	i2oDevCreate() / i2oDdmCreate()
EventCapabilities	I2O_DEV_EVENT_CAPABILITIES	i2oDevCreate() / i2oDdmCreate()
BiosInfo	I2O_DEV_BIOS_INFO	IRTOS
ChangeIndicator	I2O_DEV_CHANGE_INDICATOR	IRTOS

5.4.3.5 Assignment and Correlation of TIDs

The I$_2$O shell specification requires an IOP to use the same TIDs for each device from boot to boot. Since only the DDM knows the relation between physical devices and the I$_2$O devices that it creates via **i2oDevCreate()**, the DDM must correlate assigning TIDs to its devices. Each DDM must remember its TID assignments in non-volatile storage (using **i2oDdmMpbStore**) and associate TIDs with specific devices, so that each device gets the same TID from boot to boot.

The first time a device is created, the DDM calls **i2oDevCreate()** with a localTid of 0 in the lctInfo structure parameter. The IRTOS detects this special value and assigns the device an unused TID. The DDM learns the TID by calling **i2oDevTidGet()** and stores that TID in its MPB via **i2oDdmMpbStore()**. In the future, after a power cycle, reset, restart, or reboot, if the DDM calls **i2oDevCreate()** for that same device, it must supply the TID originally assigned by the IRTOS.

The IRTOS does, however, track the assignment of TIDs to DDMs from boot to boot. The IRTOS returns an error if a DDM attempts to create a device with a TID that was not assigned to that DDM in a previous boot.

The DDM must understand changes in physical configuration, for example, via configuration dialogues or operating principals. That is, the DDM determines whether a physical device or adapter has moved, been replaced, or been removed permanently or temporarily. By assigning the same TID, the DDM hides that the device moved or was replaced. If the device was permanently removed, the DDM releases the TID via the **i2oDdmTidRelease()** function. This makes the TID available for reassignment to other devices.

The exception is assigning the TID for DDM class devices that are created implicitly by **i2oDdmCreate()**. These TIDs are always assigned by the IRTOS, which uses the **ddmTag** parameter of **i2oDdmCreate()** to identify a DDM and assign it the same TID on each reboot. The localTid field of the lctInfo parameter of the **i2oDdmCreate()** call is ignored.

5.4.3.6 TID Table

I$_2$O messages address devices by TID, a small integer unique within a given IOP. IRTOS must map this into a pointer to a device before they can find the event queue for the request. Therefore, a TID table indexed by TID contains pointers to the corresponding devices.

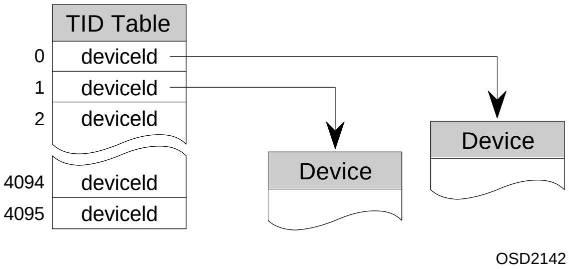


Figure 5-24 TID Table and Devices

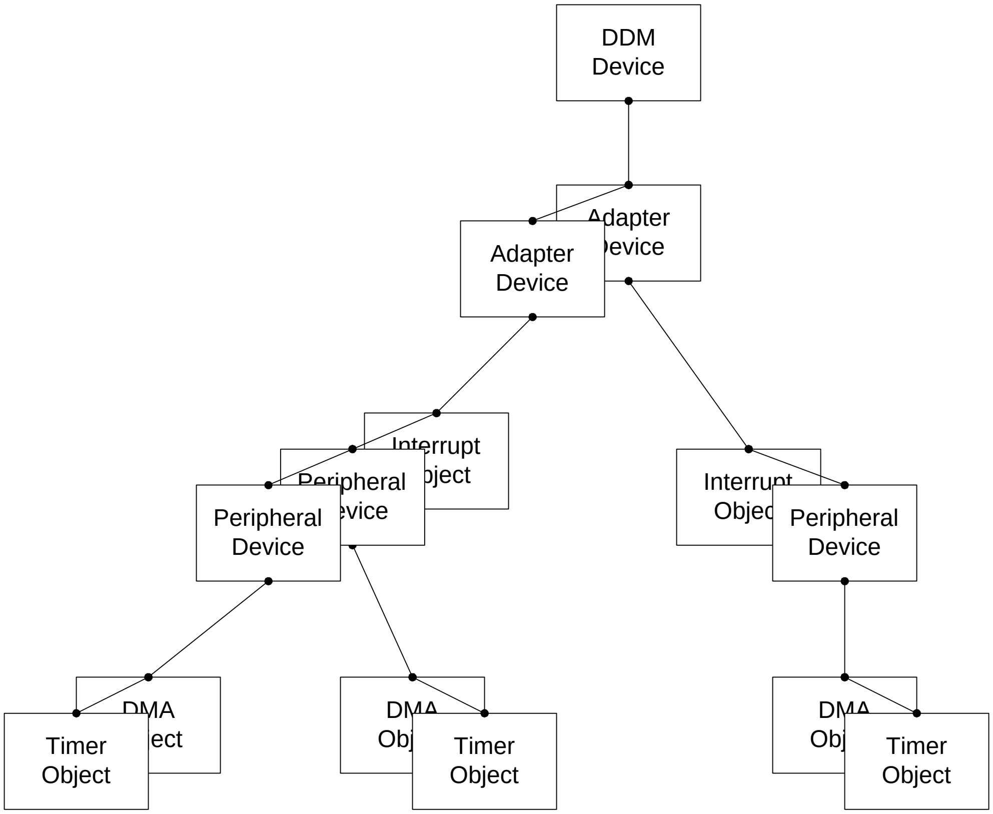
5.4.3.7 Ownership of Objects by Devices

When IRTOS objects are created, a parameter to the *create* function specifies the ID of an *owner* device object to which the each object belongs. The IRTOS uses this ownership relationship to track and reclaim every resource. When a device is destroyed, all objects it owns automatically delete as well.

Thus, a typical DDM sequence is to create a new device object and then create any other IRTOS objects required to support that device (e.g. interrupt, timer, DMA objects), all owned by that device object. IRTOS objects used by the whole DDM, not just a single device, should belong to the device object of the DDM; that is, the DDM-class device obtained by **i2oDdmDevGet()**.

A device object must have an owner, as specified in the **i2oDevCreate()** function. Often this is the device object of the DDM. However, if a DDM manages a hierarchy of devices, that hierarchy should be reflected in the ownership hierarchy. For example, a DDM for a SCSI adapter would have a device for the DDM itself, which could own all the SCSI adapter class devices created by the DDM. They, in turn, could own all the SCSI peripheral class devices created by the DDM.

The following figure below shows such a hierarchy. In this example, a DDM device owns two adapter devices: one owns two peripheral devices and the other owns one. In addition, each adapter owns an interrupt object, and each peripheral owns timer and DMA objects. Thus, deleting an adapter device automatically deletes all the peripheral devices and IRTOS objects under it.



5.4.4 Device Event Queues

Associated with each device is its event queue. Each event has a priority relative to other events. For example, hardware interrupt events might be highest priority, followed by timeout events and class request events. Events are dequeued in priority order, and FIFO within a given priority.

Although a given device is serviced by only one event queue, one event queue can service many devices. This optimizes resources when several devices would not benefit from preemptive parallel execution of events; for example, devices on a common host-bus-adapter that inherently serializes access to the devices anyway. This is described in more detail below.

5.4.5 Sharing Event Queues

A device's event queue and associated thread provide a serializing, prioritized, preemptable context for handling events for a given device. In many DDMs, several devices are accessed via a common piece of hardware, such as a host bus adapter. In this case, there is no advantage to separate event queues and threads for each device. This scenario actually poses problems, because, unless the handler functions interlock further, multiple devices might conflict by trying to simultaneously access the common hardware. Thus, a DDM often creates one event queue for several or all devices. Since each event contains the user context for its IRTOS object, events from many devices can intermingle in the event queue. Whenever the corresponding thread is free, it dequeues the highest priority event, regardless of which device it applies to. Within priority levels, events simply queue in FIFO order.

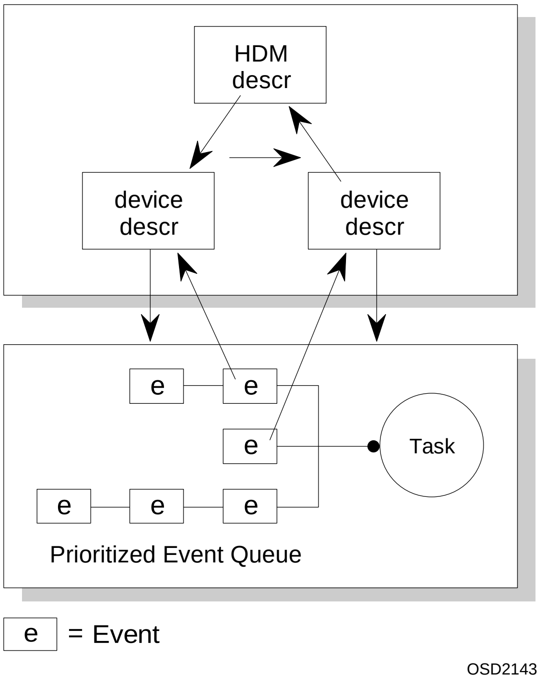


Figure 5-25 Example of Event Queue Sharing

5.4.6 Event Handler Functions and Dispatch Tables

In the IRTOS event queue facility, an event contains a pointer to a function that handles the event and two arguments from the IRTOS. When a routine posts an event, it must provide the pointer to appropriate the handler function and arguments. When certain IRTOS facilities post events, the IRTOS determines the appropriate handler in one of three ways:

1. for I_2O requests, in a dispatch table that the DDM supplies to the IRTOS, as discussed below.
2. for replies, encoded in the request's Initiator_Context field and returned in the reply, as discussed in section 5.4.7.2.
3. A driver calls certain IRTOS API functions specifying the handler for the event posted at the final conclusion of that function. See section 5.4.2.5.1.

The message dispatch table contains the event handler and queuing priority for each request message Function code for the device class (see Figure 5-26). A driver builds a message dispatch table by calling **i2oDispatchTblCreate()**. This call provides an array of entries consisting of {function code, handler entry point, event priority} triples. The IRTOS turns this array into a *dispatch table*, an internal data structure optimized for dispatching an incoming message. When the driver creates a device by calling **i2oDevCreate()**, it specifies a message dispatch table. Typically, many devices point to the same message dispatch table.

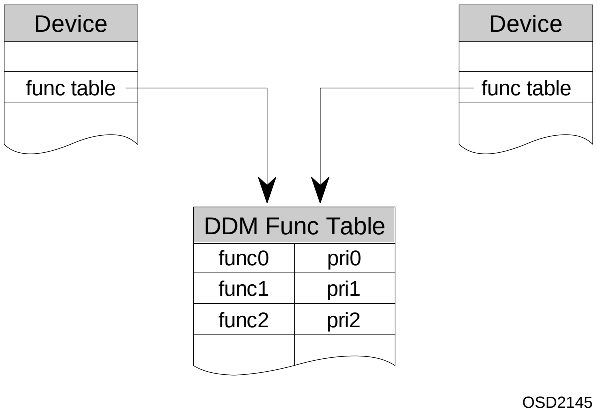


Figure 5-26 Example of Sharing a Message Dispatch Table

The message dispatcher allocates an event for each message received. The dispatcher determines the appropriate event handler. It locates message Function type in the dispatch table for the device addressed by the Target_Address. The arguments for the event are the user-specified context of the target device and a pointer to the message frame. When the event reaches the top of the queue, the event dispatcher calls the message handler with those arguments.

Table 5-33 Dispatch Table Functions

Returns	API Function Call	Description
dispatchTblId	**i2oDispatchTblCreate** `(ownerId, pFuncArray, numFuncs, &status)`	Create message handler dispatch table
Message handler declarations:		
`void`	`msgHandler (devContext, pMsgFrame)`	

Parameter	Type	Description
&status	`I2O_STATUS *`	variable to receive error code
devContext	`I2O_OBJ_CONTEXT`	user context value for device to which message is sent
dispatchTblId	`I2O_DISPATCH_TBL_ID`	ID of dispatch table
numFuncs	`I2O_COUNT`	number of entries in pFuncArray
ownerId	`I2O_OWNER_ID`	ID of owner device object, typically the DDM device object
pFuncArray	`I2O_FUNC_ENTRY *`	pointer to array of dispatch function entries: {funcCode, message handler, priority}
pMsgFrame	`I2O_FRAME`	pointer to received message

Table 5-34 Array Elements for Dispatch Functions (I2O_FUNC_ENTRY):

Parameter	Type	**Description** (each entry contains the following elements)
`funcCode`	`int`	Value to match with the Function field in a received request message
`evtHandler`	`I2O_EVENT_HANDLER *`	Pointer to the event handler that handles the message
`evtPri`	`I2O_EVENT_PRI`	Event priority level (0=highest; 7=lowest)

5.4.7 Dispatching Incoming Messages

I_2O messages come from the host (and other IOPs) via a FIFO, to which the host writes I/O request messages, implemented in IOP hardware. An I/O message arriving in the FIFO causes an interrupt in the IOP. IRTOS services this interrupt by extracting the I/O message from the FIFO. Then it dispatches the message to the appropriate event queue. The steps in dispatching the message depend on whether it is a request or a reply.

5.4.7.1 Dispatching Incoming Requests

From the received request message, the IRTOS locates the destination device. It indexes into the TID table with the TID in the Target_Address field of the request message. The TID entry points to the device and its associated dispatch table. The event handler and priority are determined from the dispatch table. The IRTOS obtains a free event from the IRTOS and creates an event. The event includes it arguments: the user context value of the target device and a pointer to the incoming message frame. The IRTOS posts the event to the device's queue at the priority indicated in the dispatch table. Figure 5-27 shows the flow of incoming requests.

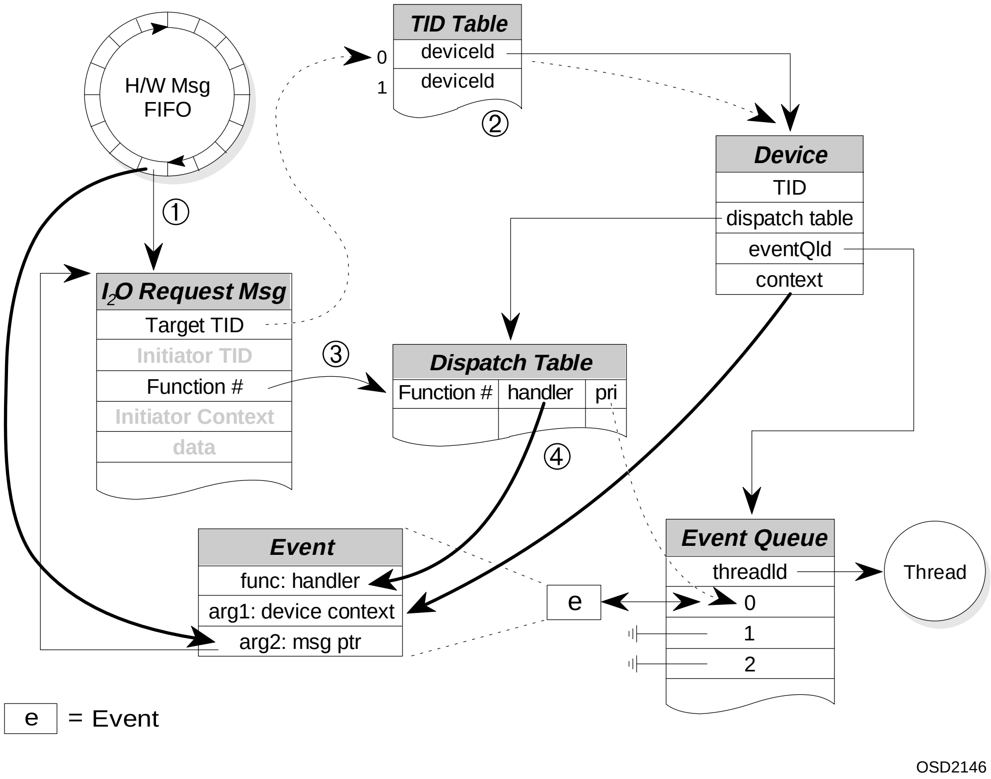


Figure 5-27 Flow of Request Message Dispatching

5.4.7.2 Dispatching Incoming Replies

The IRTOS handles replies much the way it handles requests. The IRTOS locates the destination device by indexing into the TID table with the TID in the Target_Address field of the message. From the device, the IRTOS determines the event queue and user context. Using the Initiator_Context, the IRTOS looks up the handler and the event priority. The IRTOS obtains a free event from the IRTOS and creates an event with its arguments: the user context value and a pointer to the incoming message frame. The IRTOS posts the event to the queue. Figure 5-27 shows the flow of incoming replies.

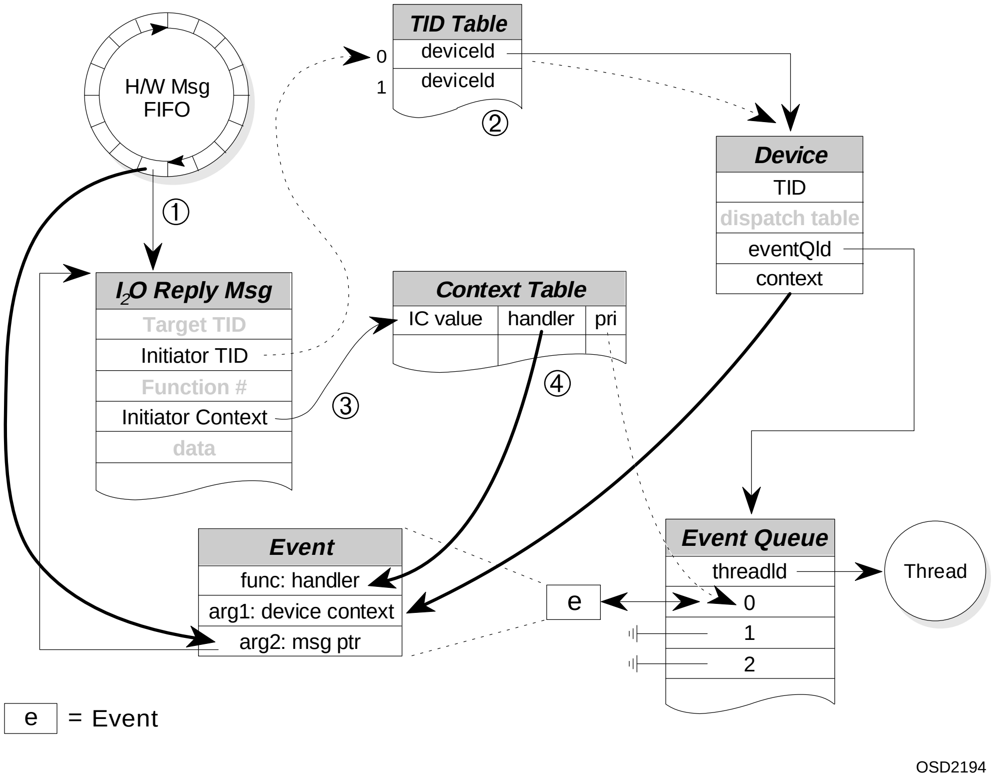


Figure 5-28 Flow of Reply Message Dispatching

5.4.8 Sending Request and Reply Messages

A DDM requests and replies via the functions in Table 5-35. DDMs must build messages, both requests and replies, in IRTOS-provided local message frames. The **i2oFrameAlloc()** function gets a local message frame in which a DDM can build a message. Note that when a message frame is allocated this way, its contents are arbitrary. Once a message is constructed in the message frame, the DDM calls **i2oFrameSend()** to send it. The message frame now belongs to the IRTOS again and the DDM must not access it further.

If the message targets a device on the local IOP, it is processed much like a message from the host:

- the device is found via the TID table
- a free event is obtained from the IRTOS' free event pool
- the message handler and priority are determined by the device message dispatch table
- the event posts to the target's queue.

If the message targets the host or another IOP, the IRTOS allocates an appropriate remote message frame on that platform, transfers the message to it, posts that remote message frame to the appropriate FIFO, and releases the original local message frame.

When a DDM receives a request or reply, it must return the incoming message frame to the IRTOS after it processes the message. The DDM can explicitly free the message frame by calling the **i2oFrameFree**() function. Alternately, the DDM can reuse the incoming message frame for an outgoing message; for example, to construct the reply to a request in the incoming message frame. This is preferable because it eliminates freeing and allocating a message frame.

The size of the message frames may vary on each I_2O messenger instance, i.e. on the host and each IOP, although all message frames must be at least 64 bytes long. The message frame provided to the DDM either in an incoming message or by a call to **i2oFrameAlloc()** is local and a size the IOP defines. However, the message frame of the destination (host or IOP) *may be smaller* than the local message frame size. Thus, the largest message a DDM can send a given TID is the smaller of the local and destination message frame sizes. The DDM can determine this maximum by calling the function **i2oFrameMaxSizeGet(tid)**. This returns the largest message frame that can be sent to the specified TID. The IRTOS returns an error if **i2oFrameSend()** is called with a frame that is too large for the destination.

Note that calling **i2oFrameMaxSizeGet()** with a TID known to be local, such as one of a DDM's own TIDs or the TID of the IRTOS Executive (i.e., 0), returns the actual size of the local message frames.

Table 5-35 I_2O Message Frame Functions

Returns	API Function Call	Description
pFrame	**i2oFrameAlloc** (&status)	Get frame for sending
void	**i2oFrameFree** (pFrame, &status)	Free frame
void	**i2oFrameSend** (pFrame, &status)	Send frame
maxFrameSize	**i2oFrameMaxSizeGet (tid)**	Get maximum frame size for sending to specified TID

Parameter	Type	Description
&status	I2O_STATUS *	Variable to receive error code
maxFrameSize	I2O_SIZE	Maximum size of frame to send to TID
pFrame	I2O_FRAME *	Pointer to message frame

When the DDM sends a request, it must provide an Initiator_Context field that the IRTOS uses when replies are received. The DDM calls the **i2oInitiatorContextBuild()** function specifying the event handler and event priority for the reply. The IRTOS returns an Initiator_Context value that the DDM can use in any frame. When a reply frame is returned, the IRTOS queues an event, just as for a request, except that, instead of using the dispatch table, the IRTOS determines the priority and event handler from the Initiator_Context.

Table 5-36 Initiator Context Function

Returns	API Function Call	Description
icValue	**i2oInitiatorContextBuild** (replyEvtHandler, replyEvtPri, &status)	Build initiator context field

Reply Message Received *evtHandler* declaration:

```
void replyEvtHandler (devContext, pFrame)
```

Parameter	Type	Description
&status	`I2O_STATUS *`	Variable to receive error code
devContext	`I2O_OBJ_CONTEXT`	Context of device specified as TID in Target_Address
icValue	`I2O_INITIATOR_CONTEXT`	Value to be specified as Initiator_Context in request message
pFrame	`I2O_FRAME *`	Pointer to reply message frame
replyEvtHandler	`I2O_EVENT_HANDLER *`	Event handler for reply message event
replyEvtPri	`I2O_EVENT_PRI`	Priority of reply message event

5.4.9 Static Messages

To create a static message, a DDM allocates a message frame by calling **i2oFrameAlloc()** and builds the message. Then it calls **i2oStaticMsgCreate()**. This instructs the IRTOS to create a static message frame using the indicated frame. If the target is on another IOP, the IRTOS communicates with that IOP to establish the static message frame. The communication with the other IOP is potentially lengthy. To avoid blocking the DDM during the inter-IOP negotiation, the **i2oStaticMsgFrameCreate()** function returns once communication with the other IOP begins. At this point, the static message frame is not fully created. After negotiating with the other IOP, the IRTOS posts an event to the DDM indicating creation of the static message frame. Now, the DDM can call **i2oStaticMsgSend()** to send the message.

Once a message frame becomes static, the DDM cannot change its content. It cannot attempt to send or free the frame using **i2oFrameSend()** or **i2oFrameFree()**. The frame is automatically freed if the static message frame is destroyed (**i2oObjDestroy()**).

Part of negotiating with the other IOP includes asking it to reserve the specified *maxOutstanding* slots in its input FIFO. This prevents FIFO overflow from repeatedly posting static message frames. Thus, the DDM must ensure that more than *maxOutstanding* number of this static message never post to the target at once.

Table 5-37 I$_2$O Static Message Functions

Returns	API Function Call	Description
staticMsgId	**i2oStaticMsgCreate** (ownerId, smContext, pFrame, maxOutstanding, evtQId, smEvtPri, smEvtHandler, &status)	Create static message
void	**i2oStaticMsgSend** (staticMsgId, &status)	Send static message

Create Static Message Frame *evtHandler* declaration:

```
void smEvtHandler (smContext, smCreateStatus)
```

Parameter	Type	Description
&status	`I2O_STATUS *`	Variable to receive error code
evtQId	`I2O_EVENT_QUEUE_ID`	ID of event queue to which static msg create event will be posted
maxOutstanding	`I2O_COUNT`	Maximum number of times static msg will be queued at one time
ownerId	`I2O_OWNER_ID`	ID of owner device object
pFrame	`I2O_FRAME *`	Pointer to message frame to use for static message
smContext	`I2O_OBJ_CONTEXT`	User context value for static msg object used in completion event
smCreateStatus	`I2O_STATUS`	Status of static message creation
smEvtHandler	`I2O_EVENT_Handler`	Event handler for static msg create event *
smEvtPri	`I2O_EVENT_PRI`	Priority of static msg create event
staticMsgId	`I2O_STATIC_MSG_ID`	ID of static message

5.4.10 Buses

Two IRTOS objects represent physical hardware entities: the Bus and Adapter objects. The adapter and the bus are hierarchically related. That is, the adapter occupies a portion of the bus space and, therefore, the driver accesses adapters via operations on that bus object.

An IOP might connect to many physical buses and DDMs may need access to address spaces on each bus. DDMs need a platform-independent way to access the various buses. Furthermore, access to a given address on a bus might require configuring certain bus windows (hardware-specific), since not all addresses of all buses can be visible at once. Thus, DDMs must access all buses through IRTOS bus objects. The IRTOS provides functions that read and write various sized data (8, 16, 32, and 64-bit) of a specified address on a specified bus. An address space argument identifies which of several different types of address spaces is accessed (e.g. memory space vs. I/O space). This address space argument is bus-specific and must correspond to the types of address space the bus type supports.

The IRTOS also provides an **i2oBusTranslate()** function that translates addresses from one bus to another. The DDM can use this, for example, to determine the address of a buffer in local or system memory as seen from another bus. That way, a device with bus master (DMA) capabilities can be programmed to read or write that buffer. Note that this function only translates addresses on the first specified bus that are actually visible from the second specified bus. If the address is not visible on the second bus, there is no legitimate translation of that address. Therefore, the function returns an error status, or takes the automatic error action, if *I2O_NO_STATUS* is specified for the status pointer.

To find the constant offset between local and system addresses (mentioned in Chapter 2 as the *base difference*), a DDM can call **i2oBusTranslate().** It translates a local address of a local buffer visible on the system bus (e.g., via the memory allocation functions) to a system address. The offset will be the difference between the two addresses. That technique applies to other buses, as well.

Every system identifies two buses: the local bus that contains IOP private memory and devices, and the system bus that hosts main system memory and system I/O devices. The

DDM obtains the busId of these buses by calling **i2oBusLocal()** and **i2oBusSystem()**, respectively.

The other bus a DDM must know about is the bus(es) where its physical adapters reside. This might be either the local bus, the system bus, or another bus the IOP can access. The DDM obtains the busId where a particular physical device resides by calling **i2oAdapterBusGet()** as described below.

Table 5-38 lists the IRTOS functions for bus objects.

NOTE

IRTOS does not provide a create *function for bus objects, because it creates all the bus objects as part of the platform-specific system initialization.*

Table 5-38 Bus Functions

Returns	API Function Call	Description
busId	**i2oBusLocal** ( )	Get ID of local bus
result8	**i2oBusRead8** (busId, busSpace, busAddr, &status)	Read byte
result16	**i2oBusRead16** (busId, busSpace, busAddr, &status)	Read 16-bit word
result32	**i2oBusRead32** (busId, busSpace, busAddr, &status)	Read 32-bit word
result64	**i2oBusRead64** (busId, busSpace, busAddr, &status)	Read 64-bit word
busId	**i2oBusSystem** ( )	Get ID of system bus
busAddr2	**i2oBusTranslate** (busId1, busSpace1, busAddr1, busId2, busSpace2, &status))	Translate a shared memory address
void	**i2oBusWrite8** (busId, busSpace, busAddr, value8, &status)	Write 8-bit word
void	**i2oBusWrite16** (busId, busSpace, busAddr, value16, &status)	Write 16-bit word
void	**i2oBusWrite32** (busId, busSpace, busAddr, value32, &status)	Write 32-bit word
void	**i2oBusWrite64** (busId, busSpace, busAddr, value64, &status)	Write 64-bit word

Parameter	Type	Description
&status	`I2O_STATUS *`	Variable to receive error code
busAddr	`I2O_ADDR32`	Address to access on bus
busAddr1	`I2O_ADDR32`	Address on bus 1
busAddr2	`I2O_ADDR32`	Equivalent address on bus 2 of specified address on bus 1
busId	`I2O_BUS_ID`	ID of bus object
busId1	`I2O_BUS_ID`	ID of bus 1
busId2	`I2O_BUS_ID`	ID of bus 2
busSpace	`I2O_BUS_SPACE`	Address space to access on bus (memory, I/O, etc.)
busSpace1	`I2O_BUS_SPACE`	Address space on bus 1
busSpace2	`I2O_BUS_SPACE`	Address space on bus 2
result8	`INT8`	8-bit value read from bus
result16	`INT16`	16-bit value read from bus
result32	`INT32`	32-bit value read from bus
result64	`INT64`	64-bit value read from bus

Parameter	Type	Description
value8	`INT8`	8-bit value written to bus
value16	`INT16`	16-bit value written to bus
value32	`INT32`	32-bit value written to bus
value64	`INT64`	64-bit value written to bus

5.4.11 Adapters

In addition to accessing buses, DDMs also need platform-independent access to specific physical devices on those buses. Therefore, the IRTOS provides an adapter object for each physical device attached to a DDM. Adapter functions provide information about the adapter, including the bus on which it resides (**i2oAdapterBusGet()**) and its physical location on that bus (**i2oAdapterPhysLocGet()**). The IRTOS also provides functions that read and write to configuration registers on the adapter in various data widths (8-, 16-, 32-, and 64-bit).

A DDM must enable and disable interrupts from an adapter. Often, this is necessary to properly synchronize activity between ISRs. ISRs run at interrupt level in response to device interrupts, and event handlers, which run in an IRTOS thread context (see section 5.4.12). Often, the DDM commands the physical device itself to enable or disable its interrupt generation. This method is the least intrusive to the rest of the system, because only interrupts from that device are affected.

However, in some cases it is impossible or undesirable to control the physical device directly in this way. In this case, one can enable and disable interrupts on the IOP itself. The functions **i2oAdapterIntLock()** and **i2oAdapterIntUnlock()** disable and re-enable interrupts from a particular adapter by locking out interrupts up to and including the interrupt level of the specified adapter. This also locks out interrupts from all other devices at or below the interrupt level of the adapter, so these calls must disable interrupts only briefly. It is preferable, however, to use **i2oAdapterIntLock()** and **i2oAdapterIntUnlock(),** rather than **i2oIntLock()** and **i2oIntUnlock()**, which disable *all* interrupts on the IOP CPU.

This specification does not limit interrupt lockout, but OS vendors can impose their own limits. These functions are provided for DDMs that absolutely need them, but are not recommended otherwise. From a marketing perspective, drivers that disable others’ interrupts for prolonged periods are considered poor performers.

A DDM receives the adapterId for a particular device in the ***DeviceAttach*** message to the DDM from the IRTOS executive.

Table 5-39 lists the IRTOS functions for adapter objects.

NOTE

IRTOS does not provide a create *call for adapter objects, because it creates all adapter objects during platform-specific initialization or bus scanning. For bus types that require the DDM to scan for its own adapters, the IRTOS creates a single adapter object.*

Table 5-39 Adapter Functions

Returns	API Function Call	Description
busId	**i2oAdapterBusGet** (adapterId, &status)	Get adapter's bus identifier
result8	**i2oAdapterConfigRead8** (adapterId, offset, &status)	Read adapter configuration register
result16	**i2oAdapterConfigRead16** (adapterId, offset, &status)	Read adapter configuration register
result32	**i2oAdapterConfigRead32** (adapterId, offset, &status)	Read adapter configuration register
result64	**i2oAdapterConfigRead64** (adapterId, offset, &status)	Read adapter configuration register
void	**i2oAdapterConfigWrite8** (adapterId, offset, value8, &status)	Write adapter configuration register
void	**i2oAdapterConfigWrite16** (adapterId, offset, value16, &status)	Write adapter configuration register
void	**i2oAdapterConfigWrite32** (adapterId, offset, value32, &status)	Write adapter configuration register
void	**i2oAdapterConfigWrite64** (adapterId, offset, value64, &status)	Write adapter configuration register
void	**i2oAdapterIntLock** (adapterId, &status)	Disable adapter interrupts
void	**i2oAdapterIntUnlock** (adapterId, &status)	Enable adapter interrupts
void	**i2oAdapterPhysLocGet** (adapterId, &physLoc, &status)	Get hardware physical location

Parameter	Type	Description
&physLoc	I2O_PHYS_LOC *	Pointer to physical location structure where location of adapter is returned (bus-specific)
&status	I2O_STATUS *	Variable to receive error code
adapterId	I2O_ADAPTER_ID	ID of adapter object
busId	I2O_BUS_ID	ID of bus on which adapter resides
offset	I2O_ADDR32	Offset of configuration register to access
result8	INT8	8-bit value read from adapter configuration register
result16	INT16	16-bit value read from adapter configuration register
result32	INT32	32-bit value read from adapter configuration register
result64	INT64	64-bit value read from adapter configuration register
value8	INT8	8-bit value written to adapter configuration register
value16	INT16	16-bit value written to adapter configuration register
value32	INT32	32-bit value written to adapter configuration register
value64	INT64	64-bit value written to adapter configuration register

5.4.12 Memory Allocation

DDMs require variable amounts of memory dynamically at run-time. Three IRTOS mechanisms allocate run-time memory with different requirements and trade- offs of memory management. Table 5-40 summarizes those memory management mechanisms. Each is described in detail in the following sections.

Table 5-40 Memory Allocation Functions Summary

Allocate Function	Free Function	Allocation Size	Allocation Frequency	Battery Backup	Typical Use
i2oMemAlloc()	**i2oMemFree()**	Variable # of bytes	Infrequent	No	Data structure
i2oPageAllocContig()	**n/a**	Contiguous pages	Initialization only	Yes	DDM managed buffers
i2oPageAlloc(), **i2oPageAllocN()**	**i2oPageFree(),** **i2oPageFreeN()**	Individual pages	Frequent	Yes	Temporary buffers

i2oMemAlloc(), I2oMemFree()

These functions allow dynamic allocation and deallocation of variable size blocks of memory. Because the algorithms for managing variable size memory are relatively slow, these functions suit relatively infrequent allocation/deallocation needs, such as creating internal data structures as devices are added. These functions are the most like the traditional **C malloc()** and **free()** functions.

i2oPageAllocContig()

This function allows a DDM a single initial block of contiguous memory whose size is determined by IRTOS based on the needs of all DDMs. This memory is permanently allocated to the device or DDM. This mechanism is typically used by drivers that want to manage their own single large extent of memory. Memory obtained from this mechanism can be recovered from a system crash if battery backup is available on the memory.

i2oPageAlloc(), i2oPageAllocN(), i2oPageFree(), i2oPageFreeN()

These functions allow dynamically allocating and deallocating fixed-size pages of memory. These functions are extremely fast. Also, because they allocate only individual, non-contiguous pages, they pose no issues of memory fragmentation. DDMs should use this allocation mechanism wherever possible and especially for frequent and/or short memory allocations. Memory obtained from this mechanism can be recovered from a system crash if battery backup is available on the memory.

5.4.12.1 Memory Attributes

IRTOS allows a DDM to specify three important characteristics of memory allocation:

access — the accessibility of the memory by various categories of external devices

cache — the cache-coherency of the memory, with respect to reading and writing to external devices battery backup whether or not the memory can be preserved by battery backup in case of a system failure (this attribute can be specified only on page allocations)

5.4.12.1.1 Access Attributes

An IOP maps different regions of local memory to external buses, so devices on those buses can read from and write to those regions. When allocating memory, a DDM must specify requirements of external devices and buses for accessing that memory. For example, a DDM allocating a buffer that will be read by a bus mastering adapter must make sure that the memory is accessible to that adapter. Table 5-41 lists the access attributes that can be specified when allocating memory.

Table 5-41 Memory Access Attributes

Access Attribute	Description
`I2O_MEM_ACCESS_PRIVATE`	Not accessible by external devices
`I2O_MEM_ACCESS_SYSTEM`	Accessible by host, and other IOPs and devices on the System bus
`I2O_MEM_ACCESS_LOCAL_ADAPTERS`	Accessible by all adapters controlled by this IOP
`I2O_MEM_ACCESS_ALL_ADAPTERS`	Accessible by all adapters in this I_2O segment
`I2O_MEM_ACCESS_BUS_SPECIFIC`	Accessible to adapters on a specific bus

PRIVATE specifies memory not accessible by any external devices. Specify this attribute for all internal data structures that are not shared with any external device.

SYSTEM specifies memory accessible from the System bus. Specify this attribute for any buffer the host will access. (This is equivalent to specifying *I2O_MEM_ACCESS_BUS_SPECIFIC* with the busId of the system bus.)

LOCAL_ADAPTERS specifies memory accessible to all devices this IOP controls. Specify this attribute for any buffer that may be accessed by a device controlled by a DDM on this IOP.

ALL_ADAPTERS specifies memory accessible to all devices in the entire I_2O segment. Specify this attribute for any buffer that may be accessed by an arbitrary I_2O device. For example, an ISM would specify this attribute for buffers that will go to lower-level DDMs without knowing the location of the adapter that may access the data. Note that the *ALL_ADAPTERS* attribute includes accessibility by the system bus if this IOP communicates peer to peer, or controls any adapters on the system bus.

BUS_SPECIFIC specifies memory accessible from a specified bus (busId provided in a separate parameter). Specify this for any buffer that will be accessed only by devices known to be the particular bus. For example, an HDM would specify this attribute for buffers to be shared with the device being controlled.

These attributes specify the *minimum* accessibility requirements of the allocated memory. The IRTOS allocates memory that has *at least* the accessibility specified, but *may* also be accessible to other devices and buses. For example, memory allocated with the *I2O_MEM_ACCESS_PRIVATE* attribute may be accessible by external devices if the IOP has no truly private memory available. To maximize protection, the IRTOS will try to provide memory as close as possible to the specified attributes.

5.4.12.1.2 Cache Attributes

When sharing local IOP memory with external devices, consider the effects of the IOP's data cache. A data cache allows cache-coherency problems, that is, the contents of memory looks different to the processor (i.e. by DDMs running on the IOP) than the external devices. IOPs solve cache-coherency different ways, perhaps disabling the cache for regions of memory or using bus snooping hardware.

A DDM must always specify what cache-coherency attributes it requires for memory being allocated. If the allocated memory will not be accessed by an external device, then the memory has no cache attribute requirements. If so, the IRTOS allocates memory with the most efficient caching available.

If the allocated memory will be written to by the processor and read by an external device, then the memory must be *safe for external reads* (*I2O_MEM_EXTERNAL_READ_SAFE*). This means that if the processor writes to the memory, then the values must be visible immediately to external reads. This occurs if that memory is uncached, write-through cached, or write-back cached with snooping, but not if the memory is write-back cached without snooping.

Similarly, if the allocated memory will be written by an external device and read by the processor, then the memory must be *safe for external writes* (*I2O_MEM_EXTERNAL_WRITE_SAFE*); that is, if an external device writes to the memory, then the values must be immediately visible to processor reads. This occurs if the memory is uncached or cached with snooping, but not if the memory is cached without snooping.

When allocating memory, the DDM specifies the cache attributes it requires for the memory by ORing together either attribute (see Table 5-42). The IRTOS allocates appropriate memory, if possible, for the specified characteristics.

A DDM should always specify the *weakest* cache-coherency requirements it can. For example, if a buffer is allocated that will be read externally but not written to externally, then the DDM should specify *only I2O_MEM_EXTERNAL_READ_SAFE.* This allows the IRTOS to allocate the buffer in write-through cached memory, rather than uncached memory. This can significantly impact the performance of the DDM.

On the other hand, one must specify *all* the cache-coherency requirements really needed. Cache-coherency problems are notoriously difficult to debug.

Table 5-42 Memory Cache Attributes

Cache Attribute	Description
`I2O_MEM_EXTERNAL_READ_SAFE`	Buffer will be read by external device
`I2O_MEM_EXTERNAL_WRITE_SAFE`	Buffer will be written by external device

For reference, Table 5-43 summarizes the cache attributes of various types of caches, although this information is irrelevant when writing a DDM since the cache attributes are sorted out by the IRTOS.

Table 5-43 Cache Attributes of Various Cache Types

Cache Type	External Read-Safe	External Write-Safe
Uncached	Yes	Yes
Write-through w/o snooping	Yes	No
Write-through w/snooping	Yes	Yes
Write-back w/o snooping	No	No
Write-back w/snooping	Yes	Yes

5.4.12.1.3 Memory Attributes of Local Message Frames

There is one important buffer whose memory attributes the DDM can*not* control: local message frames. These are obtained by DDMs either as incoming messages or by calling **i2oFrameAlloc()**. The IRTOS guarantees that these local messages will be:

1. in memory accessible to all adapters (*I2O_MEM_ACCESS_ALL_ADAPTERS*), and
2. in external read safe memory (*I2O_EXTERNAL_READ_SAFE*), but may not be external write safe. Thus, a DDM can safely set up an external device to read directly out of a local message frame (immediate data or the S/G list itself), but not to write into a local message frame.

5.4.12.2 Memory Sets

The **i2oMem…()** functions allocate and free variable length blocks of local IOP memory. Allocating variable length blocks allows dense use of available memory but incurs additional run-time overhead in managing the memory pool (e.g. finding, splitting, and recombining free blocks). Also, frequently allocating and deallocating variable length blocks can fragment the memory pool, which will decrease the size of blocks available for allocation. Therefore, these functions are intended only for infrequent allocation and freeing of memory, typically at during initialization time or changes in configuration or operating conditions. These functions are *not* intended for use by transaction basis. Instead, the page allocation facility described below should be used for such purposes.

Table 5-44 IRTOS Variable Size Memory Allocation Functions

Returns	API Function Call	Description
memSetId	**i2oMemSetCreate** (ownerId, cacheAttr, accessAttr, busId, &status)	Create a memory set
addr	**i2oMemAlloc** (memSetId, size, alignment, &status)	Allocate memory block
void	**i2oMemFree** (memSetId, addr, &status)	Free memory block

Parameter	Type	Description
&status	`I2O_STATUS *`	Variable to receive error code
accessAttr	`I2O_MEM_ACCESS_ATTR`	Access attribute of memory set (see Table 5-41)
addr	`I2O_ADDR32`	Pointer to memory block allocated via memory set
alignment	`I2O_SIZE`	Alignment requirement of allocated block
busId	`I2O_BUS_ID`	ID of bus, if accessAttr = *I2O_MEM_ACCESS_BUS_SPECIFIC*
cacheAttr	`I2O_MEM_CACHE_ATTR`	Cache attributes of memory set (see Table 5-42)
memSetId	`I2O_MEM_SET_ID`	ID of memory set object
ownerId	`I2O_OWNER_ID`	ID of owner device object
size	`I2O_SIZE`	Number of bytes to allocate via memory set

The **i2oMemSetCreate()** function creates a *memory set* object that holds the variable length memory blocks allocated via that memSet. The cacheAttr, accessAttr, and busId parameters specify the memory attributes to allocate via the memSet. The memSet object resembles a *container* for memory allocations. When a memSet object is destroyed, the memory allocations contained automatically return to the IRTOS. Thus, memSets provide the following:

1. a convenient way to specify memory attributes, and
2. they track memory allocations for automatic reclamation and clean-up.

Typically, a DDM creates, in its initialization routine and/or ATTACH message handlers, a memSet for each type of memory it needs with different sets of attributes. Depending on how it assigns memory to specific devices, it may create separate memSets for each device or collections of devices. Generally, however, creating and allocating out of fewer memSets uses system memory more efficiently.

The **i2oMemAlloc()** function allocates the specified size memory block via the specified memSet and returns the address of the allocated memory. Its address is aligned per the alignment parameter specified in the call to **i2oMemAlloc()**, i.e., if the address is evenly divisible by the value of the alignment parameter. The **i2oMemFree()** function frees a previously allocated block.

If insufficient memory is available when **i2oMemAlloc()** is called, the function returns a *NULL* address and the returned status is set to *I2O_STS_INSUFFICIENT_MEMORY*. This is *not* an error condition that invokes automatic error handling if *I2O_NO_STATUS* is specified as the status return parameter. This *out of memory* condition can occur during normal system operation and so should not cause fatal error recovery. Thus, the address returned from **i2oMemAlloc()** must *always* be tested to verify that a non-*NULL* address was returned.

5.4.12.3 Page Sets

The **i2oPage…()** functions allocate and free local IOP memory in pages. Pages are fixed size blocks of memory that are:

1. a power of two in length with a minimum of 4K bytes, and
2. *naturally aligned*, that is, on the same power of two boundary as their length.

This makes them suitable for use in scatter-gather list page elements. Since the pages are fixed-size, allocating and freeing individual pages can be quick, eliminating memory fragmentation problems. Thus, this mechanism is suited to very frequent, possibly per transaction, allocations and deallocations. Another mechanism obtains a single initial allocation of contiguous pages. Also, page sets may be battery backed-up. Table 5-45 lists the page set functions.

Table 5-45 Page Allocation Functions

Returns	API Function Call	Description
addr	**i2oPageAddrGet** (pageSetId, prevPageAddr, &status)	Get address of page in set
addr	**i2oPageAlloc** (pageSetId, &status)	Allocate page
void	**i2oPageAllocContig** (pageSetId, minPages, maxPages, pageEvtQId, pageEvtPri, pageEvtHandler, pageEvtArg, &status)	Allocate initial contiguous pages
count	**i2oPageAllocN** (pageSetId, nPages, &addrArray, &status)	Allocate pages
bbuEnabled	**i2oPageBbuEnableGet** (pageSetId, &status)	Get BBU enabled/disabled
void	**i2oPageBbuEnableSet** (pageSetId, bbuEnable, &status)	Set BBU enabled/disabled
void	**i2oPageBbuNotify** (pageSetId, evtQId, evtPri, bbuEvtHandler, bbuEvtArg, &status)	Request notification of changes in BBU status
bbuStatus	**i2oPageBbuStatus** (pageSetId, &status)	Get current BBU status
count	**i2oPageCountGet** (pageSetId, &status)	Get number of pages in set
void	**i2oPageFree** (pageSetId, addr, &status)	Free an allocate page
void	**i2oPageFreeN** (pageSetId, nPages, &addrArray, &status)	Free a list of allocated pages
pageSetId	**i2oPageSetCreate** (ownerId, pageContext, cacheAttr, accessAttr, busId, bbuAttr, &status)	Create a page set
pageSize	**i2oPageSizeGet** (pageSetId, &status)	Get size of page

pageEvtHandler and bbuEvtHandler declarations:

```
void        pageEvtHandler (pageContext, pageEvtArg)
void        bbuEvtHandler (pageContext, bbuEvtArg)
```

Parameter	Type	Description
&status	`I2O_STATUS *`	Variable to receive error code
accessAttr	`I2O_MEM_ACCESS_ATTR`	Access attribute of page set (see Table 5-41)

Parameter	Type	Description
addr	I2O_ADDR32	Pointer to page allocated via page set
bbuAttr	I2O_BBU_ATTR	Battery backup attribute of page set (see Table 5-46)
bbuEnabled	BOOL	TRUE if battery backup is enabled for this page set, FALSE if disabled
bbuEvtQId, bbuEvtPri, bbuEvtHandler, bbuEvtArg	I2O_EVENT_QUEUE_ID, I2O_EVENT_PRI, I2O_EVENT_HANDLER, I2O_ARG	Parameters for event to be posted when battery backup status changes
bbuStatus	I2O_BBU_STATUS	Current condition of battery backup (see Table 5-47)
busId	I2O_BUS_ID	ID of bus, if accessAttr = *I2O_MEM_ACCESS_BUS_SPECIFIC*
cacheAttr	I2O_MEM_CACHE_ATTR	Cache attributes of page set (see Table 5-42)
count	I2O_COUNT	Number of pages actually allocated, or in page set
list	I2O_ADDR32 *	Pointer to list of page addresses
maxPages	I2O_COUNT	Maximum number of pages desired in initial contiguous allocation (0 = as much as possible)
minPages	I2O_COUNT	Minimum number of pages required in initial contiguous allocation
nPages	I2O_COUNT	Number of pages requested to allocate or free
ownerId	I2O_OWNER_ID	ID of owner device object
pageContext	I2O_OBJ_CONTEXT	User context value for page set
pageEvtQId, pageEvtPri, pageEvtHandler, pageEvtArg	I2O_EVENT_QUEUE_ID, I2O_EVENT_PRI, I2O_EVENT_HANDLER, I2O_ARG	Parameters for event to be posted when initial contiguous allocation is complete
pageSetId	I2O_PAGE_SET_ID	ID of page set object
pageSize	I2O_SIZE	Page size in bytes
prevPageAddr	I2O_ADDR32	Address of page previously returned by **i2oPageAddrGet()** (NULL = get first page address)

The **i2oPageSetCreate()** function creates a pageSet object that holds the pages it allocates. The cacheAttr, accessAttr, busId, and bbu parameters specify the attributes of the pages that will be allocated via the pageSet. The pageSet object resembles a *container* for page allocations. A pageSet is created empty of pages.

The function **i2oPageAlloc()** adds a single page to the specified pageSet and returns the page's address. Pages are allocated from a system page pool that satisfies the attributes specified for the pageSet. The **i2oPageAllocN()** function adds the specified number of separate, non-contiguous pages to the pageSet and returns their addresses in the specified array. This is the same as calling **i2oPageAlloc()** repeatedly. In either case, the allocated pages are then owned by that pageSet. (See Figure 5-29.) Pages can be freed one at a time by calling **i2oPageFree(),** which removes it from the pageSet and returns it to the system page pool. The function **i2oPageFreeN()** frees the specified number of pages whose addresses are in the specified array. As noted, these functions are fast and may be called frequently.

If no pages are available to allocate when **i2oPageAlloc()** is called, the function returns a *NULL* address and the returned status is set to *I2O_STS_INSUFFICIENT_MEMORY*. This is *not* an error condition that invokes automatic error handling if *I2O_NO_STATUS* is specified as the status return parameter. This *out of memory* condition can occur during normal system operation and so should not cause fatal error recovery. Thus, the address returned from **i2oPageAlloc()** must *always* be tested to verify that a non-*NULL* address was returned. Similarly, **i2oPageAllocN()** returns the count of the actual number of pages allocated. The count will be less than the number requested *only* if too few pages were available to fulfill the request. Thus, the count returned must *always* be tested to find the actual number of pages allocated.

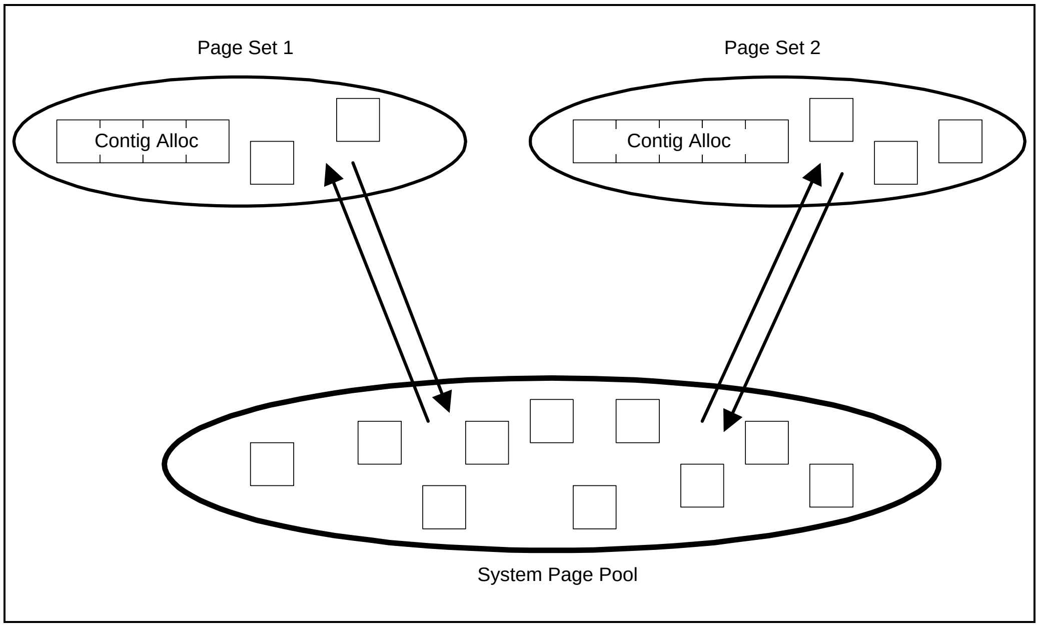


Figure 5-29: Page Sets

The pageSet facility is typically used by DDMs to allocate and free data buffers, cache buffers, and so forth, for each incoming request. In general, DDMs should allocate pages for these uses only when necessary and free them as soon as the buffer is no longer needed. Following this policy leaves the most pages available for the system to adapt to changing loads, such as demand peaks for DDMs, and configuration changes..

Typically, during initialization and/or *ATTACH* message handlers, a DDM creates a pageSet for each type of memory it needs, with different sets of attributes. Depending on how it assigns memory to specific devices, it may create separate pageSets for each device or collections of devices.

When a pageSet object is destroyed, all its pages automatically return to the IRTOS. Thus, pageSets provide the following:

1. a convenient way to specify a set of page attributes
2. tracking of a set of page allocations for automatic reclamation and clean-up, and
3. a mechanism for recovering battery backed-up pages after a system failure.

The **i2oPageSizeGet()** function returns the size of the pages in bytes.

The **i2oPageCountGet()** function returns the number of pages currently allocated to the specified pageSet. The **i2oPageAddrGet()** function returns the addresses of pages currently allocated to the specified set, one at a time. The **prevPageAddr** parameter must be specified as *NULL* on the initial call to get the first page in the set. Additional page addresses can then be obtained by calling **i2oPageAddrGet()** with **prevPageAddr** set to the address of the page returned in the previous call. The first page of the initial contiguous allocation (described below), if any, is always returned first.

5.4.12.3.1 Initial Contiguous Page Allocation

When a pageSet is created, an initial allocation of contiguous pages can be requested with **i2oPageAllocContig()** . This function requests a range of pages from minPages to maxPages. A value of 0 for maxPages indicates an unlimited maximum; that is, the DDM can use as many pages as possible. The pages are not actually allocated at the time of the call, because the IRTOS may first need to accumulate the initial contiguous page requests of other DDMs. Therefore the DDM is notified of the actual allocation by an event, as specified by pageEvtPri, pageEvtHandler, and pageEvtArg, in the **i2oPageAllocContig()** call. Once the event arrives indicating a complete allocation, the **i2oPageAddrGet()** and **i2oPageCountGet()** functions can find out the address of the pages allocated and actual number of contiguous pages allocated to the pageSet.

The **i2oPageAllocContig()** *must* run before the IRTOS receives replies to all *ADAPTERATTACH* and *DEVICEATTACH* messages, when the IRTOS makes the initial contiguous page allocations. Only a single **i2oPageAllocContig()** can be made on each page set. Call **i2oPageAllocContig()** only for contiguous pages.

5.4.12.3.2 Battery Backup

Some IOPs offer battery backup for some regions of local IOP memory. The bbuAttr parameter in the **i2oPageSetCreate()** call specifies that battery backup on the pages allocated to the pageSet is either

1. required
2. desirable but not required
3. not needed

If bbuAttr is specified as required but not available, an error will be returned. If bbuAttr is specified as desirable, but not required, then whether battery backup is available on the pageSet can be determined by the **i2oPageBbuStatus()** function described below.

Table 5-46 Battery Backup Attributes

BBU Attribute	Description
`I2O_BBU_DESIRED`	Pages should have battery backup capability, if possible
`I2O_BBU_NOT_USED`	Pages do not need battery backup capability
`I2O_BBU_REQUIRED`	Pages need battery backup capability

When the IRTOS boots after a system failure with battery-backup on, it automatically adds the recovered pages to the appropriate pageSets as they are created by the DDMs. Thus, immediately after creating a pageSet, a DDM can determine whether any battery-backed-up pages were recovered by calling **i2oPageCountGet()**. It can locate the pages by calling **i2oPageAddrGet()**. Also, the battery backup is automatically enabled for that pageSet if any battery backed-up pages were recovered (see **i2oPageBbuEnableSet()** below).

When recovering battery backed-up pages, to identify the pageSet that the pages belong to, the IRTOS uses the TID of the device specified as owning the pageSet in the **i2oPageSetCreate()** call. Therefore, each device can have only one battery backed-up pageSet for each set of memory attributes.

The **i2oPageBbuStatus()** function returns the condition of the battery backup on the specified pageSet (see Table 5-47). The **i2oPageBbuNotify()** function requests notification via an event as specified by bbuEvtPri, bbuEvtHandler, and bbuEvtArg, whenever the battery backup changes status (goes from CHARGED to UNCHARGED, or vice versa).

Table 5-47 Battery Backup Status Values

BBU Status	Description
`I2O_BBU_CHARGED`	Battery backup is present and can supply backup
`I2O_BBU_UNAVAILABLE`	No battery backup is present on this page set
`I2O_BBU_UNCHARGED`	Battery backup is present but cannot currently supply backup

The **i2oPageBbuEnableSet()** function requests the battery backup on or off for the specified page set. In implementation, the battery backup is only physically turned off if all battery backed-up page sets for all DDMs are disabled. Thus, the **i2oPageBbuEnableSet()** function tells the IRTOS that the page set contains no data (bbuEnable = FALSE) or contents that need to be preserved (bbuEnable = TRUE). This function is a no-op if the pageSet has no battery backup. Battery backup is initially off when a page set is created, unless battery backed-up pages for that pageSet were recovered from a previous system failure. If so, as noted above, the pages automatically add to the pageSet and the battery backup for that pageSet is on.

The **i2oPageBbuEnableGet()** function returns the current state of the battery backup enabling on the specified pageSet.

5.4.12.4 Persistent Memory

An I$_2$O IOP may have several types of persistent memory a DDM can use. Three distinct categories of persistent memory are supported by IRTOS APIs:

1. an IOP file store, typically implemented with FLASH memory
2. non-volatile RAM (NV-RAM), used to store faster-changing configuration and state information
3. battery-backed-up RAM, typically used to store data cache buffers

The following table summarizes the characteristics of these three types of persistent memory:

Memory Abstraction	Typical Devices	IRTOS API	Write Freq.	Typical Size	Typical Use
File store	FLASH	i2oDdmMpbStore, InstallDDM, etc.	very low	1MB	store IRTOS, DDMs, MPBs
NV-RAM	serial EPROM	i2oDevNvram…	medium	256 bytes	store faster changing configuration and state information
Battery-backup RAM	dynamic RAM w/BBU	i2oPageBbu…	high	4MB	cache buffers

The IOP file store is discussed in section 5.4.3.2. The battery-backup RAM facilities are discussed in section 5.4.12.3.2. The NV-RAM facilities are discussed below.

5.4.12.4.1 Non-Volatile RAM Allocation

Table 5-48 IRTOS NVRAM Access Functions

Returns	API Function Call	Description
size	**i2oNvramSizeGet** (devId, &status)	Get size of NV-RAM allocated to device
void	**i2oNvramSizeSet** (devId, size, &status)	Set size of NV-RAM allocated to device
void	**i2oNvramRead** (devId, addr, len, buf, &status)	Read bytes from NV-RAM
void	**i2oNvramWrite** (devId, addr, len, buf, &status)	Write bytes to NV-RAM

Parameter	Type	Description
&status	`I2O_STATUS *`	Variable to receive error code
addr	`I2O_ADDR32`	Byte offset into NVRAM segment to read or write
buf	`I2O_ADDR32`	Address of buffer to be written to or read from NVRAM
devId	`I2O_DEV_ID`	ID of device object that owns the NVRAM segment
len	`I2O_SIZE`	Number of bytes of NVRAM segment to read or write
size	`I2O_SIZE`	Number of bytes of NVRAM to allocate to device

Some I_2O platforms have a small amount (typically a few hundred bytes) of non-volatile RAM (NV-RAM) available for DDMs. This NV-RAM is typically implemented with a serial EPROM or similar device. It generally can be written to fairly often, as opposed to the FLASH-based IOP file store, which can be written to only during significant configuration changes. A DDM may use NV-RAM to store current state information. Often, this is used with battery-backed-up RAM to recover critical state and data after a system failure.

The model of NV-RAM in the IRTOS API defines each device, identified by TID, with a single segment of NV-RAM. Initially, the NV-RAM segment for a new device is of 0 length. The segment is associated with the TID across reboots of the IOP, so that the data written to a device's NV-RAM segment remains in the same segment after a reboot. The NV-RAM segment assigned to a device is automatically freed when the TID associated with that device is released.

The **i2oDevNvramSizeSet()** function sets the size of the NV-RAM segment assigned to the device. This function may fail if insufficient NV-RAM is available to satisfy the request. If the new size exceeds the old, the segment is extended without affecting its original contents.

The contents of the new additional bytes of NV-RAM is unspecified. If the new size is smaller than the old, the segment is truncated and the remainder of the old segment is lost.

Note that in actual implementation, changing the size of a segment may involve copying and moving segments around. However, as far as the application is concerned, the segment simply grows or is truncated as described above.

The **i2oDevNvramSizeGet()** function gets the size of the NV-RAM segment assigned to the device. This size is 0 if the device never allocated a segment previously using **i2oDevNvramSizeSet()**.

The NV-RAM segment assigned to a device is accessed by the DDM with the **i2oDevNvramRead()** and **i2oDevNvramWrite()** functions. These functions allow data to be read or written to the NV-RAM segment starting at the specified address in the segment (each segment starting at address 0) and extending the specified length. If the access exceeds the length of the segment, the call returns an error (or takes the appropriate error handling action).

5.4.13 Interrupt Objects

DDMs that actually control hardware devices (HDMs) must respond to their interrupts. An IRTOS responds by creating an interrupt object. A device creates as many interrupt objects as it needs, i.e., one for each physical interrupt to which it responds.

Table 5-49 IRTOS Interrupt Handling Functions

Returns	API Function Call	Description
intId	**i2oIntCreate** (ownerId, intContext, adapterId, isrHandler, isrArg, evtQId, maxEvts, &status)	Create interrupt object
void	**i2oIntEventPost** (intId, evtPri, intEvtHandler, intEvtArg, &status)	Send event from interrupt object
inIsr	**i2oIntInIsr** ()	Determine if currently in ISR
key	**i2oIntLock** ()	Lock interrupts at kernel lock level
void	**i2oIntUnlock** (key)	Restore previous interrupt level

Interrupt ISR and evtHandler declarations:

```
intHandled = isrHandler (intContext, isrArg)
void         intEvtHandler (intContext, intEvtArg)
```

Parameter	Type	Description
&status	`I2O_STATUS *`	Variable to receive error code
adapterId	`I2O_ADAPTER_ID`	ID of adapter that generates interrupt
evtPri	`I2O_EVENT_PRI`	Priority of interrupt event
evtQId	`I2O_EVENT_QUEUE_ID`	ID of event queue to which IRTOS posts interrupt events
inIsr	`BOOL`	TRUE = executing in ISR thread context, FALSE = executing in event queue thread context
intContext	`I2O_OBJ_CONTEXT`	User context value for interrupt object
intEvtArg	`I2O_ARG`	User value to be passed to **intEvtHandler()**
intEvtHandler	`I2O_EVENT_HANDLER`	Event handler for interrupt event

Parameter	Type	Description
		*
intHandled	`BOOL`	TRUE = interrupt was handled by this ISR, FALSE = interrupt not from this device
intId	`I2O_INT_ID`	ID of interrupt object
isrArg	`I2O_ARG`	User value to be passed to **isrHandler()**
isrHandler	`I2O_ISR_HANDLER *`	Interrupt handler function that processes interrupts from device
key	`I2O_INT_LOCK_KEY`	Opaque value representing previous interrupt lock state; value returned by **intLock()** must be supplied to **intUnlock()**
maxEvts	`I2O_COUNT`	Maximum number of interrupt events that can be pending at one time
ownerId	`I2O_OWNER_ID`	ID of owner device object

In the **i2oIntCreate()** call, adapterId identifies the source of the interrupt. isrHandler is the address of an ISR. The IRTOS calls **isrHandler()** at interrupt level when a hardware interrupt from the specified adapter occurs. isrArg is an argument passed in **isrHandler()**. intEvtHandler is the address of a DDM event handler function the IRTOS invokes at the normal DDM thread level when the ISR posts an interrupt event. The maxEvts parameter the driver limit the number of outstanding interrupt events in the queue.

Thus a driver responds to device interrupts at two levels: directly at interrupt level in an ISR, and via an event handler executed by the driver thread. One should minimize processing at interrupt level.

Because the interrupt vectors can be multiplexed (a requirement of the PCI bus, for example), the **isrHandler()** function must probe its device to determine the source of the interrupt. If its device does not have an interrupt pending, **isrHandler()** returns *FALSE* to indicate that it did not handle the interrupt. If **isrHandler()** discovers that its device is interrupting, then it returns *TRUE* to indicate that it did handle the interrupt. It can then do minimal response-critical processing of the interrupt. If the interrupt is a level-sensitive signal, then the ISR must clear or disable its source.

The ISR can also post an event to the DDM thread-level by calling **i2oIntEventPost()**. Note that **isrHandler()** need not post an event on every interrupt, but only when thread-level processing by the DDM is required. The IRTOS posts the interrupt event only if the number of outstanding interrupt events is less than the maxEvts parameter of the **i2oIntCreate()** function. Typically, this parameter is set to 1, because a single service call to **intEvtHandler()** can process all outstanding interrupt tasks.

The **isrHandler()** might look like:

```
BOOL isrHandler (MY_STRUCT* intContext, int isrArg)
    {
    ... probe device ...
    if (device is not interrupting)
        return (FALSE);                 /* interrupt not handled */

    . .. handle device's minimal, immediate interrupt needs ...
    if (device needs longer term attention)
        i2oIntEventPost (intContext->intId, intEvtPri, intEvtHandler,
                          intEvtArg, NO_STATUS);
        return (TRUE);                     /* interrupt handled */
        }
```

intEvtHandler()is called with the user context of the interrupt object that posted the event and the intEvtArg value specified in the **i2oIntEventPost()** call. This intEvtArg is an arbitrary four-byte value, as far as the system is concerned; it is solely for communication between **isrHandler()** and **intEvtHandler()**. Thus **intEvtHandler()** is declared as:

```
void intEvtHandler (I2O_CONTEXT intContext, int intEvtArg)
```

5.4.13.1 Calling IRTOS Functions in an ISR

The IRTOS invokes **isrHandler()** at interrupt level in a special interrupt handling context of the IOP. Thus **isrHandler()** is *not* executing in normal IRTOS thread context. Therefore many IRTOS functions are not callable from the ISR function, nor from any function called by an ISR function. The table below lists the IRTOS functions that *are* callable from ISRs.

A program determines if it is executing in the special ISR context or a normal thread context by calling the function **i2oIntInIsr()**. This function returns *TRUE* if called from an ISR context and *FALSE* if called from a normal thread context.

Table 5-50 IRTOS Functions That Can Be Called From an ISR

API Function Call
i2oAdapterBusGet()
i2oAdapterConfigRead...
i2oAdapterConfigWrite...
i2oAdapterIntLock()
i2oAdapterIntUnlock()
i2oAdapterPhysLocGet()
i2oBusLocal()
i2oBusRead…
i2oBusSystem()
i2oBusTranslate()
i2oBusWrite…
i2oBusyWait()
i2oErrorAction()
i2oIntEventPost()
i2oIntInIsr()
i2oIntLock()
i2oIntUnlock()
i2oObjContextGet()
i2oSemGive()

5.4.13.2 Controlling IOP Interrupts

The functions **i2oIntLock()** and **i2oIntUnlock()** disable and re-enable, respectively, interrupts on the IOP CPU. This guarantees that sections of code can run without interruption. This has significant impact on the rest of the IOP, since the normal response of the IRTOS and other DDMs is completely inhibited. These functions should be used only for short durations and as a last resort.

The IRTOS allow synchronizing and mutually excluding programs. The **i2oAdapterIntLock()** and **i2oAdapterIntUnlock()** functions are slightly less intrusive than the alternatives, because they disable interrupts only up to the level of a specific adapter. For synchronization between thread-level code, three types of semaphores have minimal impact on other programs in the IOP.

5.4.14 Timer Objects

DDMs must often provide timeouts on operations. In IRTOS, DDMs create *timer objects* for various timing needs. After the DDM creates a timer object, it starts the timer by specifying the timeout’s duration and event handler function. When the specified number of microseconds elapse, the event is queued, which invokes the specified event handler. A timer can be canceled at any time. Only one event can be outstanding for any given timer object.

Since the timer functions use an unsigned 32-bit integer to specify the microsecond values, the maximum duration is limited. The table below details the maximum duration from 32-bit timer with a granularity of 1 *usec*. For longer durations, invoke a single timer repeatedly.

Table 5-51 Maximum Timer Duration

4,294,967,295	microseconds
4,295	seconds
72	minutes

Table 5-52 lists the IRTOS functions for timer objects.

Table 5-52 IRTOS Timer Functions

Returns	API Function Call	Description
void	**i2oTimerCancel** (timerId, &status)	Cancel timer
timerId	**i2oTimerCreate** (ownerId, timerContext, evtQId, &status)	Create timer object
usecs	**i2oTimerElapsed** (timerId, &count, &status)	Get time elapsed since start of timer
usecs	**i2oTimerEventRes** ( )	Get resolution of event timing
void	**i2oTimerRepeat** (timerId, usec, evtPri, timerEvtHandler, timerEvtArg, &status)	Start periodic timer
usecs	**i2oTimerStampRes** ( )	Get resolution of time stamping
void	**i2oTimerStart** (timerId, usec, evtPri, timerEvtHandler, timerEvtArg, &status)	Start one-shot timer

Declaration of Timer *evtHandler* :

```
void evtHandler (timerContext, evtArg)
```

Parameter	Type	Description
&count	`I2O_COUNT *`	Pointer of variable to receive count of expirations of period timer
&status	`I2O_STATUS *`	Variable to receive error code
evtPri	`I2O_EVENT_PRI`	Priority of timer event
evtQId	`I2O_EVENT_QUEUE_ID`	ID of event queue to post timer events to
ownerId	`I2O_OWNER_ID`	ID of owner device object
timerContext	`I2O_OBJ_CONTEXT`	User context value for timer object
timerEvtArg	`I2O_ARG`	User value to be passed to timerEvtHandler()
timerEvtHandler	`I2O_EVENT_HANDLER`	Event handler for timer event *
timerId	`I2O_TIMER_ID`	ID of timer object
usecs	`I2O_USECS`	Number of microseconds

To use the IRTOS timer facility, a timer object must first be created by **i2oTimerCreate()**. Start the timer by calling either **i2oTimerStart()** or **i2oTimerRepeat()**. **i2oTimerStart()** initiates a one-shot timer that expires when *at least* the specified number of microseconds elapse. Upon expiration, the specified event is posted and the timer idles. **i2oTimerRepeat()** initiates a periodic timer that resets after each expiration. On each expiration, the event is posted. If the event is still posted (not yet been handled) from a previous expiration, the periodic timer continues to run, but further expirations are not posted while the event remains queued. Once the event dequeues, the next timer expiration posts another event. The count returned by **i2oTimerElapsed()**, described below, always reflects the actual number of expirations.

An unlimited number of timer objects can be created in an IRTOS system, even though only a small number of hardware timers are usually available (typically 1 to 3). For this reason, the timer objects are generally implemented as a software queue driven by a periodic hardware timer. This is known as the platform timer. The period of the platform timer determines the resolution of timer expirations, since the IRTOS can discover an expiration only upon interruption from the platform timer. Thus, the timer object guarantees that *at least* the specified number of microseconds have elapsed when the expiration event is posted. The actual time elapsed before the event is posted can be considerably longer, depending on the period of the platform timer (100 Hz is typical, e.g. 10,000 usecs). Calling **i2oTimerEvtRes()** provides the resolution of timer objects.

Calling **i2oTimerElapsed()** provides timestamping by reading the number of microseconds that elapsed since the timer started. With a periodic timer, **i2oTimerElapsed()** also returns a count of the number of expirations since the timer started, in addition to the number of microseconds since the last expiration.

Unlike timer expirations, the timestamping is generally very high resolution because it involves only reading a high frequency counter. The resolution of the timestamping function is obtained by calling **i2oTimerStampRes()**.

For general-purpose timestamping without event posting, the driver specifies the platform timer directly, without creating a timer object, by supplying a NULL timerId to **i2oTimerElapsed()**. In this case, the function returns a count of the system timer interrupts (or *ticks*) since the system started and the number of microseconds since the last system timer interrupt, i.e., 1 tick = **i2oTimerEvtRes()**.

5.4.15 DMA Objects

IOPs may have several DMA channels, each with different attributes or capabilities. The IRTOS DMA objects allow device drivers to queue DMA operations with specific attributes. Each DMA operation is then executed by a DMA channel that can satisfy the request as soon as such a channel is available. When a DMA operation completes, or gets an error or timeout, an event is posted to the DDM.

Table 5-53 lists the IRTOS functions for DMA objects.

Table 5-53 IRTOS DMA Functions

Returns	API Function Call	Description
void	**i2oDmaCancel** (dmaId, mode, xferContext, &status)	Cancel DMA transfer
dmaId	**i2oDmaCreate** (ownerId, busId1, busSpace1, busId2, busSpace2, maxXfers, createFlags, evtQId, evtPri, &status)	Create DMA object
void	**i2oDmaResume** (dmaId, &status)	Resume DMA after error
reqStatus	**i2oDmaXfer** (dmaId, addr1, addr2, length, xferFlags, xferContext, dmaEvtHandler, &status)	Do DMA transfer of single buffer
reqStatus	**i2oDmaXferFrag** (dmaId, list1, offset1, list2, offset2, maxBytes, xferFlags, xferContext, dmaEvtHandler, &status)	Do DMA transfer of scatter-gather list fragment
reqStatus	**i2oDmaXferList** (dmaId, list1, list2, xferFlags, xferContext, dmaEvtHandler, &status)	Do DMA transfer of scatter-gather list
Declaration of *evtHandler* for DMA Object :		
void	dmaEvtHandler (xferContext, dmaStatus)	

Parameter	Type	Description
&status	I2O_STATUS *	Variable to receive error code
addr1	I2O_ADDR32	Starting address on bus 1
addr2	I2O_ADDR32	Starting address on bus 2
busId1	I2O_BUS_ID	Local ID of source bus 1 to transfer from
busId2	I2O_BUS_ID	Local ID of destination bus 2 to transfer to
busSpace1	I2O_BUS_SPACE	Address space on source bus 1
busSpace2	I2O_BUS_SPACE	Address space on destination bus 2
cancelMode	I2O_CANCEL_MODE	Mode of cancel operation (see Table 5-58)
createFlags	I2O_DMA_CREATE_FLAGS	DMA object option flags (see Table 5-54)
dmaEvtHandler	I2O_EVENT_HANDLER *	Event handler for DMA completion event
dmaId	I2O_DMA_ID	ID of DMA object
dmaStatus	I2O_STATUS	Status of DMA completion (see Table 5-56)
evtPri	I2O_EVENT_PRI	Priority of DMA completion events
evtQId	I2O_EVENT_QUEUE_ID	ID of event queue to post DMA events to
length	I2O_SIZE	Number of bytes to transfer
list1	I2O_SG_ELEMENT *	Pointer to scatter-gather list for bus 1
list2	I2O_SG_ELEMENT *	Pointer to scatter-gather list for bus 2
maxBytes	I2O_SIZE	Maximum number of bytes of S/G list to transfer
maxXfers	I2O_COUNT	Maximum number of transfers outstanding
offset1	I2O_SIZE	Number of bytes to skip in list1
offset2	I2O_SIZE	Number of bytes to skip in list2
ownerId	I2O_OWNER_ID	ID of owner device object
reqStatus	I2O_STATUS	Status of DMA request (see Table 5-55)
xferContext	I2O_OBJ_CONTEXT	User context value for DMA transfer

Parameter	Type	Description
xferFlags	I2O_DMA_XFER_FLAGS	Transfer option flags (see **Table 5-57 DMA Transfer Flag Values**)

5.4.15.1 Creating DMA Objects

To use the DMA facilities, a driver first creates a DMA object by calling **i2oDmaCreate()**. This DMA object is like a virtual DMA channel between the two buses (busId1 and busId2) specified in the create call. If the buses support multiple address spaces (e.g. memory, i/o, configuration) then the bus' address spaces (busSpace1 and busSpace2) are also identified in the create call.

Memory is pre-allocated to hold a maximum of maxXfers simultaneous outstanding DMA transfers. Note that each transfer may require about 50 bytes of memory, depending on IRTOS implementation, so a DDM should minimize the maximum number of transfers a DMA object is created with.

The create call also specifies the event queue ID to which events post upon DMA completion and the event priority where they will queue.

Several options can be specified for the DMA object by ORing flags in the createFlags parameter. Normally, the IRTOS is free to queue requested DMA transfers to physical DMA engines as soon as they are available. Under certain circumstances, this may cause transfers in parallel or even out of the order in which they were requested. However, if a DMA object is created with the *I2O_DMA_SERIAL_MODE* option, then all transfers queued to that object execute strictly in the order they are requested and no transfer starts before all previous transfers queued to the same object complete. (The only exception to the serial order is *local-to-local* transfers; see 5.4.15.8.) Note that this *serial transfers* mode also changes how DMA errors and timeouts are handled (see 0).

Note that there is never any implied temporal ordering of requests on *different* DMA objects.

The *I2O_DMA_BUS_1_DEMAND_MODE* and *I2O_DMA_BUS_2_DEMAND_MODE* flags indicate that the DMA to the specified bus should occur in *demand mode*, which drives transfers by a device on that bus. These options require platform-specific support from both the IOP hardware and the external devices, and thus are available only in certain configurations.

Table 5-54 DMA Creation Flags Values

Flag	Description
I2O_DMA_BUS_1_DEMAND_MODE	set bus 1 in demand mode
I2O_DMA_BUS_2_DEMAND_MODE	set bus 2 in demand mode
I2O_DMA_SERIAL_MODE	execute all transfers strictly in order requested

5.4.15.2 Requesting DMA Transfers

Three functions provide DMA transfers via a DMA object:

i2oDmaXfer() requests a DMA transfer between simple buffers on the two buses, specifying the start address on each bus (addr1 and addr2) and the length of the transfer (length).

i2oDmaXferList() requests a DMA transfer between scatter-gather lists on the two busses (list1 and list2). The length of the transfer is determined by the shorter of the two S/G lists.

i2oDmaXferFrag() requests a DMA transfer between fragments of S/G lists. The arguments are as in the **i2oDmaXferList()** function, except that an offset is specified for each list (offset1 and offset2), indicating the number of bytes to skip in each list, and the maximum length (maxBytes). The actual length is the lesser of the length of either the list fragment or the specified maximum length.

Note that in the **i2oDmaXferList()** and **i2oDmaXferFrag()** functions, *the scatter-gather lists must not be modified until the transfer is complete.*

These functions ignore the *direction* bits embedded in the list elements (see Chapter 3). These bits indicate to the message transport whether the scatter-gather list elements are input or output, but the DMA transfer functions ignore them.

All three transfer request functions return a request status. That status is *I2O_STS_OK* if the transfer queues and *I2O_STS_DMA_FULL* if the transfer cannot queue because the DMA object already has the maximum number of outstanding requests. The maximum is specified when the DMA object is created.

I2O_STS_DMA_FULL is not considered a fatal error since it may happen in the normal course of using a DMA object. Therefore, it is returned separately as the result of the transfer request functions, rather than in the usual status return (&status). Therefore, it invokes no automatic error actions.

Table 5-55 DMA Request Status Values

Flag	Description
I2O_STS_OK	transfer queued successfully
I2O_STS_DMA_FULL	transfer not queued because DMA object already has maximum outstanding requests

5.4.15.3 DMA Completion Events

In all of the **i2oDmaXfer…()** functions, the caller supplies two parameters for an event that will be posted after the transfer (unless the *I2O_DMA_NO_EVENT* flag is specified as described below): the handler function (evtHandler), and a *context* value (xferContext), usually a pointer to an internal data structure.

When the transfer completes or terminates with an error or timeout, an event posts that will call the specified function, with the xferContext as the first argument, and the dmaStatus of the transfer as the second argument.

Table 5-56 DMA Completion Status Values

Flag	Description
`I2O_STS_OK`	transfer completed successfully
`I2O_STS_DMA_ERROR`	transfer failed
`I2O_STS_DMA_TIMEOUT`	transfer timed out

5.4.15.4 DMA Transfer Options

In all of the **i2oDmaXfer…()** functions, the caller supplies a xferFlags word that specifies transfer options.

One flag specifies the direction of the transfer. By default, transfers progress from bus 1 to bus 2. By specifying the *I2O_DMA_DIR_REVERSE* option, the transfer reverses, so that bus 2 becomes the source and bus 1 the destination.

Another flag allows optimizing posting completion events. By default, a completion event is posted when each transfer completes. However, a driver may not need notification of some transfers. In this case, some overhead can be saved by specifying the *I2O_DMA_NO_EVENT* option when a transfer is queued. This suppresses posting the transfer completion event, and automatically makes room for another transfer. A completion event posts only if an error or timeout occurs on that transfer.

For example, if a given I/O request to a driver generates five DMA transfers, and the driver cannot process further until all five complete, then the *I2O_DMA_NO_EVENT* could be specified on the first four, leaving only the last transfer to post a completion event. This saves 80% (four out of five) of the overhead of the DMA interrupts and completion event posting and handling. However, this optimization is possible *only* if the DMA object is created in *serial transfers* mode. Otherwise, completing the fifth transfer does *not* imply completion of the first four.

As another example, if a device is automatically triggered by transferring bytes or by accessing an address, then the driver might know implicitly that the DMA transfer completed when it receives a device interrupt. In this case, the driver can eliminate all DMA interrupts and completion events (other than errors) by specifying the *I2O_DMA_NO_EVENT* option on every transfer. Note that this optimization applies even in normal *non-serial* mode.

For **i2oDmaXferList()** and **i2oDmaXferFrag()**functions, two more flags, I2O_DMA_SRC_SGL_CONTEXT_64 and I2O_DMA_DST_SGL_CONTEXT_64, respectively indicate that the default transaction context size for the source and destination scatter-gather lists is 64-bit. Use of these flags removes the need to place an attribute element at the start of lists indicating 64-bit transaction context size. Should an attribute element be present in the source or destination list, it overrides any usage of these flags.

Table 5-57 DMA Transfer Flag Values

Flag	Description
`I2O_DMA_DIR_REVERSE`	DMA from bus 2 to bus 1
`I2O_DMA_NO_EVENT`	Do not post event when transfer completes
`I2O_DMA_SRC_SGL_CONTEXT_64`	Indicates transaction context size for the source scatter-gather list defaults to 64-bit.
`I2O_DMA_DST_SGL_CONTEXT_64`	Indicates transaction context size for the destination scatter-gather list defaults to 64-bit.

5.4.15.5 DMA Error Handling

If a DMA error occurs on any transfer, a completion event posts, regardless of whether or not *I2O_DMA_NO_EVENT* was specified for the transfer in error. The dmaStatus argument to the completion handler indicates an error, *I2O_STS_DMA_ERROR*.

In addition, the IRTOS keeps timers on all active DMA channels and automatically detects a stalled DMA transfer. In this case, a completion event is posted, again regardless of whether or not *I2O_DMA_NO_EVENT* was specified, with a dmaStatus argument of *I2O_STS_DMA_TIMEOUT*.

In both cases, the DMA transfer that incurred the error or timeout is automatically canceled and removed from the DMA queue.

Normally, one transfer error or timeout does not affect any other transfers on that same object, and normal operation of the DMA object continues. However, if the DMA object was created in the serial-transfers mode (*I2O_DMA_SERIAL_MODE*), then any DMA error or timeout suspends all DMA transfers on that object. However, it does not prevent queuing additional transfers to the DMA object with the **i2oDmaXfer…()** functions. To restart transfers after an error or timeout, a DDM *must* call **i2oDmaResume()**. Typically, a DDM will want to cancel other transfers related to the one that failed, if any, and then resume the DMA. It is unnecessary to cancel the transfer that incurred the error since it will already cancel automatically. It is an error to call **i2oDmaResume()** if the DMA object is *not* suspended due to an error or timeout.

5.4.15.6 Canceling Transfers

Queued DMA transfers can be canceled before completion using **i2oDmaCancel()**. This function cancels either transactions with the specified xferContext, or all transfers on the specified DMA object, depending on the specified mode. Note that a canceled transfer may not have started at all, be in progress and partially complete, or have completed but its event not dequeued yet. In any case, the transfer is discarded, **i2oDmaCancel()** is called, and no completion event is received for a canceled transfer.

Table 5-58 DMA Cancel Modes

Value	Description
`I2O_DMA_CANCEL_MATCHING`	cancel transfers with the specified context
`I2O_DMA_CANCEL_ALL`	cancel all transfers

5.4.15.7 Interpreting System Bus Addresses

When the DMA creation specifies the system bus, the addresses in the transfer for that bus may actually reside on the IOP local bus, on the primary bus, or on a secondary bus bridged to the primary. The **i2oDmaXfer…()** functions check the range for the actual location of the addresses and invoke the appropriate DMA engines. Or they do local memory copies as described below.

5.4.15.8 Local-to-local Transfers

A requested DMA transfer may be entirely or partly local-to-local, without the driver knowing. A source or destination fragment of a DMA transfer is local if:

a) the corresponding bus was specified as the local bus in the **i2oDmaCreate()** call,

b) the corresponding bus was specified as the system bus in the **i2oDmaCreate()** call, and the specified system address maps to local memory,

c) the fragment is an element of a scatter-gather list, marked as a local address,

d) the fragment is an immediate data element in a scatter-gather list.

As they set up the DMA transfer, the **i2oDmaXfer…()** functions detect any part of a transfer between two local fragments and translate it into a local-to-local memory copy done in-line, rather than by a DMA engine. Completion events post as usual.

Local-to-local transfers are an exception to the serial order dictated by the serial-transfer mode (*I2O_DMA_SERIAL_MODE*), since these transfers are in-line during the request processing. However, this does not affect the strictly serial execution of the non-local-to-local fragments in this mode. Furthermore, in serial transfers mode, the completion event of a local-to-local transfer does not post until all transfers before the local-to-local transfer finish. Thus, the behavior of completion events is unaffected by local-to-local transfers, even in serial transfers mode.

5.4.16 Threads

When an event queue is created with **i2oEventQCreate()**, a thread simply loops, dequeuing events and invoking the specified handler function for each. This event queue thread therefore provides the context for all DDM functions.

It is not usually necessary for a DDM to create any threads besides the one implicitly created when an event queue is created. However, some I/O subsystems may be more complex than a single DDM. Advanced I/O facilities requiring a more sophisticated software architecture may use a set of cooperating threads. Additional threads can be created by calling **i2oThreadCreate()**. The parameters to this call specify the thread priority, options, stack size, initial entry point, and initial argument to the entry point.

The ID of the thread that services a particular event queue is provided by **i2oEventQThreadGet().i2oThreadIdSelf()** provides the ID of the current running thread. Also, specifying a NULL for the **threadID** parameter in any thread control function implies the current thread, or *self*.

A thread can be suspended and relinquish control to other threads for a specified amount of time by calling **i2oThreadDelay()**.

The default error action for the thread is set by calling **i2oThreadErrorActionSet()**. When the thread calls an IRTOS function with the *status* pointer argument specified as *I2O_NO_STATUS*, an error occurs, and the error action specified in **i2oThreadErrorActionSet()** is invoked.

IRTOS uses a priority-based preemptive scheduling algorithm. This means it always runs the highest priority thread that is ready. Thread priorities are assigned from 0 to 255 with 0 highest priority and 255 the lowest. A thread is assigned a priority when it is created. A thread's priority can be obtained by **i2oThreadPriGet()** and set by **i2oThreadPriSet()**.

It is possible to disable the normal IRTOS scheduling whenever a thread runs by calling **i2oThreadLock()**. Any thread that has disabled IRTOS scheduling this way is *not* preempted by higher-priority threads. Only when such a thread blocks is another thread scheduled. Normal IRTOS scheduling is re-enabled by calling **i2oThreadUnlock()**. Because disabling the thread scheduling destroys the normal prioritized response of the system, preemption should rarely be locked and only briefly.

Table 5-59 lists the IRTOS functions for threads.

Table 5-59 IRTOS Thread Functions

Returns	API Function Call	Description
threadId	**i2oThreadCreate** (ownerId, threadPri, threadOptions, threadStackSize, threadInitFunc, threadArg, &status)	Create thread
void	**i2oThreadDelay** (usecs, &status)	Delay thread for specified usecs
errorAction	**i2oThreadErrorActionGet** (threadId, &userErrorHandler, &status)	Get error action for thread
void	**i2oThreadErrorActionSet** (threadId, errorAction, userErrorHandler, &status)	Set error action for thread
threadId	**i2oThreadIdSelf** (&status)	Get threadId of current thread
void	**i2oThreadLock** (&status)	Disable preemption of current thread
threadPri	**i2oThreadPriGet** (threadId, &status)	Get priority of thread
void	**i2oThreadPriSet** (threadId, threadPri, &status)	Set priority of thread
void	**i2oThreadUnlock** (&status)	Enable preemption of current thread

Parameter	Type	Description
&status	`I2O_STATUS *`	Variable to receive error code
errorAction	`I2O_ERROR_ACTION`	Response if error encountered
ownerId	`I2O_OWNER_ID`	ID of owner device object
threadArg	`I2O_ARG`	User value to pass to **threadInitFunc**()
threadId	`I2O_THREAD_ID`	ID of thread object
threadInitFunc	`I2O_THREAD_FUNC *`	Initial entry point of thread
threadOptions	`I2O_THREAD_OPTIONS`	Thread options - none defined at this time
threadPri	`I2O_THREAD_PRI`	Priority of thread: 0 (highest) - 255 (lowest)
threadStackSize	`I2O_SIZE`	Size of thread's stack in bytes
usecs	`I2O_USECS`	Number of microseconds to delay thread
userErrorHandler	`I2O_ERROR_HANDLER`	User error handler to call if errorActio*n = **I2O_ERR_ACT_USER**

Table 5-60 IRTOS Thread Options

Value	Description
`I2O_THREAD_OPTS_NONE`	No thread options are currently defined; this argument is a placeholder for future IRTOS facilities

5.4.17 Busy Wait

The **i2oBusyWait()** function accommodates short, timing-critical delays some hardware requires. This function spins the processor in a busy loop until *at least* the specified number of microseconds elapse. Obviously, since this function is wasting CPU cycles, it should be used only when absolutely necessary for very short durations. This function can be called from both the interrupt and thread levels. This function does not disable thread preemption or interrupts, so if necessary, complete those steps before calling this function.

Table 5-61 IRTOS Busy Wait Function

Returns	API Function Call	Description
`void`	**i2oBusyWait** (usecs, &status)	Busy wait for specified usecs

Parameter	Type	Description
`usecs`	`I2O_USECS`	number of microseconds to busy wait
`&status`	`I2O_STATUS *`	variable to receive error code

5.4.18 Semaphores

The IRTOS environment supports all the synchronization a simple DDM requires. However, a DDM may need to synchronize explicitly with other DDMs or other threads that are created explicitly. The basic IRTOS mechanism for thread synchronization and mutual exclusion is semaphores. Three types of semaphores are provided:

- binary semaphores
- counting semaphores
- mutual exclusion (mutex) semaphores

5.4.18.1 Binary Semaphores

A binary semaphore is a simple flag that is available (full) or unavailable (empty). When a thread takes a binary semaphore, using **i2oSemTake()**, the outcome depends on whether the semaphore is available or unavailable. If the semaphore is available, then it becomes unavailable and the function returns immediately. If the semaphore is unavailable, then the action depends on the timeout specified in the call:

1) If the timeout is specified as *I2O_WAIT_FOREVER*, the thread blocks and waits indefinitely to obtain the semaphore.

2) If the timeout is specified as *I2O_NO_WAIT*, the function returns immediately with the error condition *I2O_STS_UNAVAILABLE.*

3) If the timeout is specified as something other than *I2O_WAIT_FOREVER* or *I2O_NO_WAIT*, the thread blocks for the timeout specified. If the semaphore is still unavailable when the timeout expires, the function returns with the error condition *I2O_STS_TIMEOUT.*

When a thread gives a binary semaphore, using **i2oSemGive()**, the outcome also depends on the whether the semaphore is available or unavailable at the time of the call. If the semaphore is already available, giving the semaphore has no effect at all. If the semaphore is unavailable and no thread waits to take it, then the semaphore is simply made available. If the semaphore is unavailable and one or more threads were pending its availability, then the first thread blocked on that semaphore is unblocked, and the semaphore is left unavailable.

Binary semaphores are very fast with low overhead, and are useful for both thread synchronization and mutual exclusion. When used for thread synchronization, one thread waits for the synchronization by taking a semaphore from another thread. When used for mutual exclusion, a binary semaphore is associated with a resource that needs to be guarded. When a thread wants to access the resource, it takes the associated binary semaphore, preventing any other thread’s access to the resource at the same time. When the thread finishes accessing the guarded resource, it releases the resource by giving the semaphore, allowing other threads to take the semaphore and gain access to the resource.

5.4.18.2 Counting Semaphores

Counting semaphores are another means to synchronize threads and use mutual exclusion. The counting semaphore works like the binary semaphore, except that it tracks the number of times a semaphore is given. Every time a semaphore is given, the count increments; every time a semaphore is taken, the count decrements. When the count reaches zero, a thread that tries to take the semaphore is blocked. As with the binary semaphore, if a semaphore is given and a thread is already blocked waiting on that semaphore, that thread becomes unblocked. However, if a semaphore is given and no threads blocked, then the count is simply incremented. This means that a counting semaphore given twice can be taken twice without blocking, in contrast to a binary semaphore.

Counting semaphores can guard multiple instances of a resource. For example, the use of five tape drives can be coordinated using a counting semaphore with an initial count of five, or a FIFO with 256 entries can be implemented using a counting semaphore with an initial count of 256. The initial count is specified as an argument to the **i2oSemCCreate()** function.

5.4.18.3 Mutual Exclusion Semaphores

A mutual exclusion (mutex) semaphore is a specialized binary semaphore designed to address issues inherent in mutual exclusion, including priority inversion, deletion safety, and recursive access to resources.

The behavior of the mutex semaphore is the same as a binary semaphore, with the following exceptions:

- It can only be used for mutual exclusion.
- It can be given only by the thread that took it.
- The same thread can take it repeatedly and it is not available until that thread gives the semaphore the same number of times it took it. (This allows nested taking of the same semaphore).
- It can optionally prevent deleting threads while they own the semaphore. (This provides deletion safety).
- It can optionally invoke the priority inheritance protocol when a thread blocks on a semaphore that is currently owned by a lower-priority thread. This prevents priority inversion, which blocks indefinitely a high-priority thread with an unrelated lower-priority thread.

Table 5-62 lists the IRTOS functions for semaphores.

Table 5-62 IRTOS Semaphore Functions

Returns	API Function Call	Description
semId	**i2oSemBCreate** (ownerId, semOptions, initialState, &status)	Create binary semaphore
semId	**i2oSemCCreate** (ownerId, semOptions, initialCount, &status)	Create counting semaphore
semId	**i2oSemMCreate** (ownerId, semOptions, &status)	Create mutex semaphore
void	**i2oSemTake** (semId, timeout, &status)	Take semaphore
void	**i2oSemGive** (semId, &status)	Give semaphore

Parameter	Type	Description
&status	I2O_STATUS *	Variable to receive error code
initialCount	I2O_COUNT	Initial count of counting semaphore
initialState	I2O_SEM_B_STATE	Initial state of semaphore object: *I2O_SEM_FULL* = semaphore is available *I2O_SEM_EMPTY* = semaphore no available
ownerId	I2O_OWNER_ID	ID of owner device object
semId	I2O_SEM_ID	ID of semaphore object
semOptions	I2O_SEM_OPTIONS	Semaphore options (see Table 5-63)

Parameter	Type	Description
timeout	`I2O_USECS`	Number of microseconds before canceling **i2oSemTake()**; *I2O_NO_WAIT* = immediate return if not available, *I2O_WAIT_FOREVER* = no timeout

Table 5-63 IRTOS Semaphore Options

Value	Description
`I2O_SEM_OPT_Q_PRIORITY`	Waiting threads queued in thread priority order
`I2O_SEM_OPT_Q_FIFO`	Waiting threads queued in FIFO order
`I2O_SEM_OPT_DELETE_SAFE`	Prevent deletion of thread holding mutex (mutex semaphore only)
`I2O_SEM_OPT_INVERSION_SAFE`	Use priority inheritance protocol (mutex semaphore only)

5.4.19 Pipes

While semaphores provide a fast way to synchronize and interlock threads, often a higher-level mechanism is necessary to allow cooperating threads to communicate with each other. In IRTOS, the primary inter-thread communication mechanism within a single IOP is a *pipe.*

A variable number of messages, each of variable length, can be sent to a pipe. The maximum number of messages and the their maximum lengths are specified when a pipe is created, and enough memory is pre-allocated to accommodate that much data (number of messages x maximum length) in the pipe. Messages are sent to a pipe by calling the **i2oPipeSend()** function and messages are received from a pipe by calling **i2oPipeReceive()**.

Note that the messages transported using pipes do not affect formal I$_2$O messages that use the I$_2$O Message Transport. The I$_2$O Message Transport sends via **i2oFrameSend()** and receives via events posted to event queues; it contents are defined by the I$_2$O Shell specification. Rather, pipe messages are simple local buffers of data whose contents are completely up to the user.

When a thread calls **i2oPipeSend()** there are several possible outcomes:

1) If there is free space for the message in the pipe, then the message queues to the pipe and the function returns immediately.

2) If no space is available for the message and the timeout is specified as *I2O_WAIT_FOREVER*, the thread blocks and waits indefinitely for free space to queue the message to the pipe.

3) If no space is available for the message and the timeout is specified as *I2O_NO_WAIT*, the function returns immediately with the error condition *I2O_STS_UNAVAILABLE.*

4) If no space is available for the message and the timeout is specified as something other than *I2O_WAIT_FOREVER* or *I2O_NO_WAIT*, the thread blocks for the timeout specified. If no space is available to queue the message when the timeout expires, the function returns with the error condition *I2O_STS_TIMEOUT*.

Similarly, when a thread calls **i2oPipeReceive()** there are several possible outcomes:

1) If messages are already queued to the pipe, then the first message dequeues into the caller’s buffer and the function returns immediately.

2) If no message is available and the timeout is specified as *I2O_WAIT_FOREVER*, the thread blocks and waits indefinitely for a message on the pipe.

3) If no message is available and the timeout is specified as *I2O_NO_WAIT*, the function returns immediately with the error condition *I2O_STS_UNAVAILABLE*.

4) If no message is available and the timeout is specified as other than *I2O_WAIT_FOREVER* or *I2O_NO_WAIT*, the thread blocks for the timeout specified. If no message arrives before the timeout expires, the function returns with the error condition *I2O_STS_TIMEOUT*.

In any case, the length of the message received is returned as the value of the function, or as 0 if no message is returned.

Table 5-64 lists the IRTOS functions for pipes.

Table 5-64 IRTOS Pipe Functions

Returns	API Function Call	Description
pipeId	**i2oPipeCreate** (ownerId, maxMsgs, maxMsgLen, pipeOptions, &status)	Create pipe
void	**i2oPipeSend** (pipeId, buf, nBytes, pri, timeout, &status)	Send buffer to pipe
nBytes	**i2oPipeReceive** (pipeId, buf, maxBytes, timeout, &status)	Receive buffer from pipe

Parameter	Type	Description
&status	`I2O_STATUS *`	Variable to receive error code
buf	`I2O_ADDR32`	Pointer to message buffer
maxBytes	`I2O_SIZE`	Maximum number of bytes of message to receive
maxMsgLen	`I2O_SIZE`	Maximum length of message that can be queued to this pipe
maxMsgs	`I2O_COUNT`	Maximum messages that can be queued to this pipe
nBytes	`I2O_SIZE`	Number of bytes in message
ownerId	`I2O_OWNER_ID`	ID of owner device object
pipeId	`I2O_PIPE_ID`	ID of pipe object
pipeOptions	`I2O_PIPE_OPTIONS`	Pipe options (see Table 5-65)
pri	`I2O_PIPE_PRI`	Message queuing priority: *I2O_PIPE_PRI_NORMAL* = queue message at tail of pipe queue *I2O_PIPE_PRI_URGENT* = queue message at head of pipe queue
timeout	`I2O_USECS`	Number of microseconds before canceling send or receive; *I2O_NO_WAIT* = immediate return if not available *I2O_WAIT_FOREVER* = no timeout

Table 5-65 IRTOS Pipe Options

Value	Description
I2O_PIPE_OPT_Q_PRIORITY	Waiting threads queued in thread priority order
I2O_PIPE_OPT_Q_FIFO	Waiting threads queued in FIFO order

5.4.20 IOP Information Functions

Two functions give DDMs configuration information about the IOP on which they are running.

Table 5-66 IOP Information Functions

Returns	API Function Call	Description
pConfigInfo	**i2oIopConfigGet** ( )	Get pointer to IOP configuration information
isLocal	**i2oIopTidIsLocal** (tid, &status)	Determine if TID is local

Parameter	Type	Description
&status	`I2O_STATUS *`	Variable to receive error code
isLocal	`BOOL`	TRUE if TID is on this IOP, FALSE otherwise
pConfigInfo	`I2O_IOP_CONFIG_INFO *`	Pointer to configuration info for this IOP
tid	`I2O_TID`	TID to test

Table 5-67 IOP Configuration Info Structure

Member	Type	Description
configInfoSize	`I2O_SIZE`	Length of configuration info structure
i2oVersion	`I2O_COUNT`	I$_2$O version supported by this IOP
iopFlags	`I2O_IOP_CONFIG_FLAGS`	32-bit/64-bit mode
inboundFrameSize	`I2O_SIZE`	Size of message frames on this IOP
sysPageSize	`I2O_SIZE`	Size of pages on this I$_2$O segment
bitBucketAddr	`I2O_ADDR32`	Address of a bit bucket
bitBucketSize	`I2O_SIZE`	Size of bit bucket

The function **i2oIopConfigGet()** returns a pointer to a static structure defining various configuration constants for the IOP. The IOP configuration information contains:

configInfoSize The total number of bytes in the configuration information structure.

i2oVersion The version number of the I$_2$O specification supported by this IOP

iopFlags Bit-specific fields that indicate the IOP’s current modes and capabilities.

Bits 1,0; ContextFieldSizeCapability

0,0 Supports only 32-bit context fields.

0,1 Supports only 64-bit context fields.

1,0 Supports 32-bit & 64-bit context fields, but not concurrently.

1,1 Supports 32-bit & 64-bit context fields concurrently.

Bits 3,2; CurrentContextFieldSize

0,0	not configured.
0,1	32-bit context fields only.
1,0	64-bit context fields only.
1,1	both 32-bit or 64-bit context fields concurrently.

inboundFrameSize The size in bytes of this IOP's inbound message frames.

sysPageSize The size in bytes of pages on this I_2O segment.

bitBucketAddr The address of a *bit bucket* buffer where any DDM can send unneeded data. For example, this buffer can be the destination for DMA or external bus mastered transfers when it is necessary or more efficient to transfer and discard data. In general, the IRTOS supplies a bit bucket the size of one host page (sysPageSize bytes), but it may be smaller if the host page size is inappropriate. The actual size is specified by the bitBucketSize field. *However*, if this address is NULL, then the IRTOS is not supplying a bit bucket. A DDM must allocate its own, or use an algorithm not requiring a bit bucket.

bitBucketSize The size in bytes of the bit bucket described above.

The **i2oIopTidIsLocal()** function returns TRUE if the specified TID is on the same IOP as the caller, and FALSE otherwise. This may be used by DDMs that want to optimize algorithms differently for local versus remote targets.

5.4.21 ANSI C Library

The ANSI Standard C Library (ANSI X3.159) is intended to provide a standard base of commonly-used C functions. Many C programmers know these functions and use them routinely in programs. To help develop I_2O DDMs and other IOP software, and to increase the portability of code to and from the IOP, the IRTOS API includes the functions from the ANSI C library that pertain to developing IOP software. IOP software developers can rely on these functions for any DDM loaded into an IOP that complies with the I_2O core specification.

Some ANSI C functions are not re-entrant in a multithreaded environment, because they return pointers to strings or structures that are often implemented as static or global buffers (e.g. *div()*). In these cases, the proposed POSIX 1003.1b specification provides alternate functions that are re-entrant. These functions require the caller to pass the buffer or structure in which the result is returned. These functions receive the same name as the ANSI functions with **_r** appended (e.g. *div_r()*). Because the IRTOS environment is multithreaded, the re-entrant POSIX functions are provided, but the original ANSI functions are not.

Below are the ANSI C functions included in and excluded from the I_2O API.

Table 5-68 ANSI Standard C Functions Included in IRTOS

Standard	Category	Returns	Function	Description
ansi	stdio	len	`sprintf (buf, fmt, args…)`	Format a string to a buffer
ansi	stdio	count	`sscanf (buf, fmt, args…)`	Scan values from string in a buffer
ansi	stdlib	int	`abs (value)`	Compute absolute value of int
posix	stdlib	void	`div_r (numer, denom, &divStruct)`	Quotient and remainder of int
ansi	stdlib	long	`labs (value)`	Compute absolute value of long
posix	stdlib	void	`ldiv_r (numer, denom, &divStruct)`	Quotient and remainder of long
ansi	stdlib	int	`rand ()`	Generate pseudo-random integer
ansi	stdlib	void	`srand (seed)`	Set the seed for *rand()*
ansi	string	ptr	`memchr (buf, chr, len)`	Search buffer for char
ansi	string	int	`memcmp (p1, p2, len)`	Compare two buffers
ansi	string	ptr	`memcpy (pDest, pSrc, len)`	Copy a buffer to a non-overlapping buffer
ansi	string	ptr	`memmove (pDest, pSrc, len)`	Copy a buffer to a possibly overlapping buffer
ansi	string	ptr	`memset (buf, chr, len)`	Fill a buffer
ansi	string	ptr	`strcat (dest, append)`	Concatenate strings
ansi	string	ptr	`strchr (str, chr)`	Find first occurrence of char
ansi	string	int	`strcmp (str1, str2)`	Compare strings

Table 5-68 ANSI Standard C Functions Included in IRTOS (continued)

Standard	Category	Returns	Function	Description
ansi	string	ptr	`strcpy (dest, src)`	Copy strings
ansi	string	length	`strcspn (str, chrset)`	Find length to next character in set
ansi	string	length	`strlen (str)`	Get length of string
ansi	string	ptr	`strncat (dest, append)`	Concatenate strings with limit
ansi	string	int	`strncmp (str1, str2)`	Compare strings with limit
ansi	string	ptr	`strncpy (dest, src)`	Copy strings with limit
ansi	string	ptr	`strpbrk (str, chrset)`	Find first occurrence of char set
ansi	string	ptr	`strrchr (str, chr)`	Find last occurrence of char
ansi	string	length	`strspn (str, chrset)`	Find length to next char not in set
ansi	string	ptr	`strstr (str, substr)`	Find occurrence of substring

Table 5-69 ANSI Standard C Functions Not Included in IRTOS

`abort`	`asctime_r`	`atan2`	`atoi`	`calloc`	`clock`
`acos`	`asin`	`atexit`	`atol`	`ceil`	`cos`
`asctime`	`atan`	`atod`	`bsearch`	`clearerr`	`cosh`

- ctime
- ctime_r
- difftime
- div
- exit
- exp
- fabs
- fclose
- fdopen
- feof
- ferror
- fflush
- fgetc
- fgetpos
- fgets
- fileno
- floor
- fmod
- fopen
- fprintf
- fputc
- fputs
- fread
- free
- freopen
- frexp
- fscanf
- fseek
- fsetpos
- ftell
- fwrite
- getc
- getchar
- getenv
- gets
- gmtime
- gmtime_r
- isalnum
- isalpha
- iscntrl
- isdigit
- isgraph
- islower
- isprint
- ispunct
- isspace
- isupper
- isxdigit
- ldexp
- ldiv
- localeconv
- localtime
- localtime_r
- log
- log10
- longjmp
- malloc
- mblen
- mbstowcs
- mbtowc
- mktime
- modf
- perror
- pow
- printf
- putc
- putchar
- puts
- qsort
- raise
- realloc
- remove
- rename
- rewind
- scanf
- setbuf
- setjmp
- setlocale
- setvbuf
- signal
- sin
- sinh
- sqrt
- strcoll
- strerror
- strerror_r
- strftime
- strtok
- strtok_r
- strtol
- strtoul
- strxfrm
- system
- tan
- tanh
- time
- tmpfile
- tmpnam
- tolower
- toupper
- ungetc
- va_arg
- va_end
- va_start
- vfprintf
- vprintf
- vsprintf
- wcstombs
- wctomb

5.4.22 64-Bit Integer Arithmetic

The IRTOS API includes 64-bit integer arithmetic functions.

5.4.23 Floating Point Arithmetic

IRTOS does not support floating point arithmetic. IOP CPUs are not required to provide floating point instructions and the IRTOS environment is not required to emulate floating point. Therefore, depending on the compiler and the specific IOP platform, any floating point numbers or functions in a DDM can cause either a load error when the DDM downloads to the IOP, or a program exception at run time.

5.4.24 Configuration Support

A DDM must support device management and configuration mechanisms defined in Chapter 3 - Managing I_2O Devices via the ***UtilParamsGet***, ***UtilParamsSet*** and ***UtilConfigDialog*** messages. The IRTOS provides the following functions to automate processing of these messages. These functions simplify the implementation of a DDM's configuration support.

Table 5-70 Configuration Functions

Returns	API Function Call	Description
void	**i2oCfgParamMsgReply** (devId, context, &msg, numGroups, &groupArray, &eventClients, &status)	Process a ***UtilParamsGet*** or ***UtilParamsSet*** message and issue a reply message.
`void`	**i2oCfgDialogMsgReply** (devId, context, setNumber, &msg, numGroups, &groupArray, &eventClients, &status)	Process a ***UtilConfigDialog*** message and issue a reply message.

Parameter	Type	Description
`devId`	`I2O_DEV_ID`	Device ID of caller.
`context`	`I2O_OBJ_CONTEXT`	User context value.
`setNumber`	`I2O_COUNT`	Set number from which TCL scripts should be read (see 5.3.3.2, Module Script Table).
`&msg`	`I2O_MESSAGE_FRAME*`	Pointer to ***UtilConfigDialog***, ***UtilParamsGet*** or ***UtilParamsSet*** message frame.
`numGroups`	`I2O_COUNT`	Count of parameter groups defined in `&groupArray`.
`&groupArray`	`I2O_PARAMS_GROUP_DEF*`	Pointer to array of parameter group definitions (see 5.4.24.1 Group Declarations)
`&eventClients`	`I2O_EVENT_CLIENT_LIST*`	Pointer to a list of clients currently registered for event notification (5.4.24.2).
`&status`	`I2O_STATUS *`	Variable to receive error code.

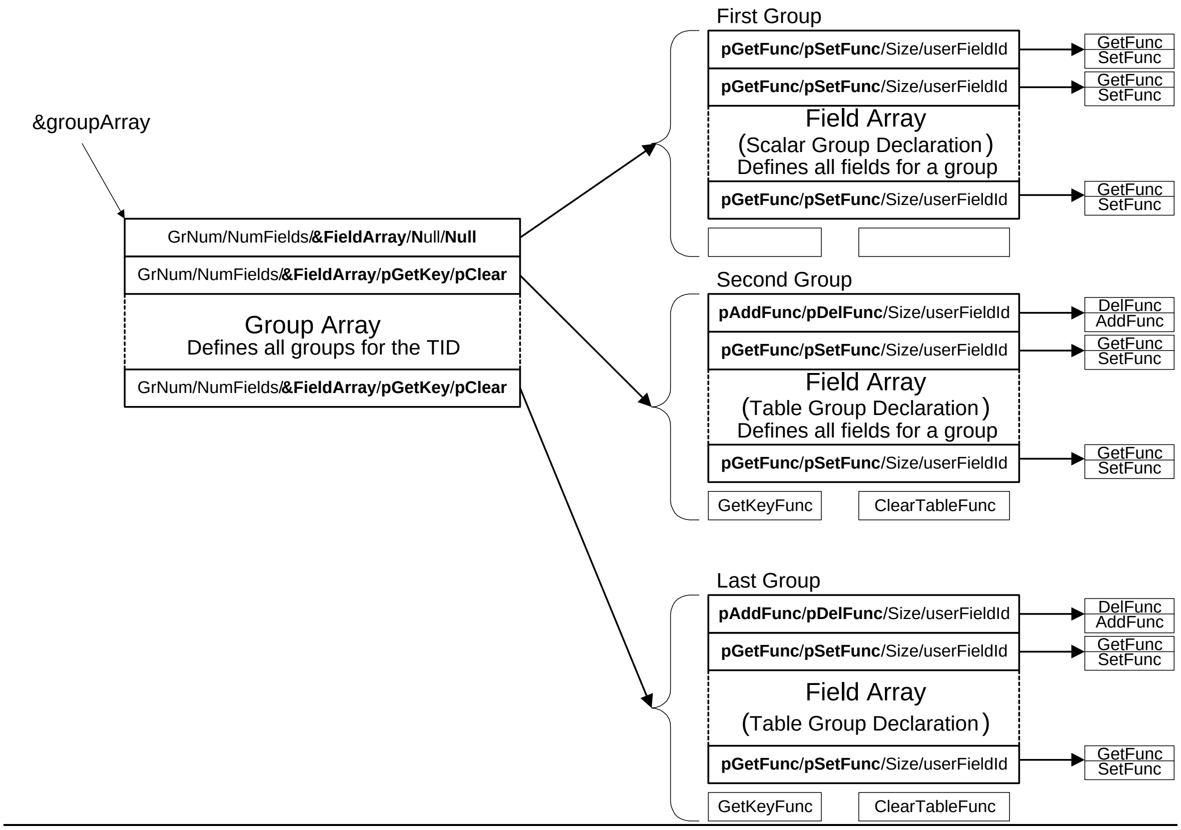
A DDM must not make any accesses to a message frame once it has passed that frame via the **i2oCfgParamMsgReply()** or **i2oCfgDialogMsgReply()** function. IRTOS frees the frame once it completes processing.

Processing of ***UtilParamsGet***, ***UtilParamsSet*** and ***UtilConfigDialog*** messages generally involve several DMA transfers. Calls to **i2oCfgParamMsgReply()** or **i2oCfgDialogMsgReply()** typically return before processing completes. Completion of the operation continues in the background transparent to the DDM as DMA transfers complete.

In order to utilize these IRTOS configuration functions, a DDM must define its configuration data using the standardized data structures described in the following sections.

5.4.24.1 Group Declarations

The DDM identifies its parameter groups in an array, with each structure in the array providing the declaration of a parameter group. Each parameter group declaration provides an array of field declarations with each structure in the field array providing the declaration of a field. Figure 5-30 illustrates this hierarchy. The DDM passes a pointer to the group array in the **i2oCfgParamMsgReply()** and **i2oCfgDialogMsgReply()** function calls.



&groupArray
GrNum/NumFields/&FieldArray/Null/Null
GrNum/NumFields/&FieldArray/pGetKey/pClear
Group Array
Defines all groups for the TID
GrNum/NumFields/&FieldArray/pGetKey/pClear
First Group
pGetFunc/pSetFunc/Size/userFieldId
pGetFunc/pSetFunc/Size/userFieldId
Field Array
(Scalar Group Declaration)
Defines all fields for a group
pGetFunc/pSetFunc/Size/userFieldId
GetFunc
SetFunc
Second Group
pAddFunc/pDelFunc/Size/userFieldId
pGetFunc/pSetFunc/Size/userFieldId
Field Array
(Table Group Declaration)
Defines all fields for a group
pGetFunc/pSetFunc/Size/userFieldId
DelFunc
AddFunc
GetKeyFunc
ClearTableFunc
Last Group
pAddFunc/pDelFunc/Size/userFieldId
pGetFunc/pSetFunc/Size/userFieldId
Field Array
(Table Group Declaration)
pGetFunc/pSetFunc/Size/userFieldId
GetKeyFunc
ClearTableFunc

Figure 5-30. Hierarchy of Parameter Group Declaration

The Group Array contains a number of structures defined in Table 5-71.

Table 5-71 Parameter Group Declartion Structure

Member	Type	Description
groupNumber	`U16`	16-bit identifier for the group.
FieldCount	`U16`	Number of fields in the group (number of entries in fieldArray).
FieldArray	`I2O_PARAMS_FIELD_DEF *`	An array defining the fields of the group (see 5.4.24.1.1.1and 5.4.24.1.1.2).
getKeysFunction	`I2O_PARAMS_GET_KEYS_FUNC`	Pointer to function that obtains key field values for rows currently in the group (see 5.4.24.1.2).
clearFunction	`I2O_PARAMS_CLEAR_FUNC`	Pointer to function that deletes all rows from the group (see 5.4.24.1.3).

A typical parameter group array definition might be:

```
I2O_PARAMS_GROUP_DEF parameterGroups [] =
    {
      { 0x8001, 5, group8001FieldArray, getKeysFunc0, clearFunc0 },
      { 0x8002, 1, group8002FieldArray, getKeysFunc1, clearFunc1 },
      { 0x8003, 2, group8003FieldArray, getKeysFunc2, clearFunc2 },
      …
    };
```

The getKeysFunction and clearFunction members, described in the following sections, are only meaningful to table groups. Scalar groups must define both these members as NULL. For table groups, the getKeysFunction must be implemented as described below. The clearFunction can be defined as NULL if the table group cannot be cleared via the configuration interface.

5.4.24.1.1 Field Array

The DDM specifies the fields of each parameter group as an array of field declarations. The DDM specifies each field within a group in terms of *get* and *set* functions, rather than any actual data storage location.

The field array is a set of field declarations structures based on the structure specified in Table 5-72.

Table 5-72 Basic Field Declaration Structure

Member	Type	Description
pSetFunction	`I2O_PARAMS_FIELD_ACCESS_FUNC`	Pointer to function that writes field value
pGetFunction	`I2O_PARAMS_FIELD_ACCESS_FUNC`	Pointer to function that reads field value
fieldSize	`I2O_COUNT`	Field size in bytes
userFieldId	`U32`	Arbitrary value associated with field.

The pSetFunction and pGetFunction components identify the functions provided by the DDM that the IRTOS calls when it access the field. The IRTOS passes the field’s userFieldId value as an argument to the set or get function. The DDM can use the userFieldId in various ways. It allows a single function to serve as the set or get function for several fields.

5.4.24.1.1.1 Scalar Group Field Array

The DDM specifies a scalar group as an array of basic field declarations. A typical scalar group definition might be:

```
I2O_PARAMS_FIELD_DEF scalarGroup [] =
    {
       { pSetFunction0, pGetFunction0, fieldSize0, userFieldId0 },
       { pSetFunction1, pGetFunction1, fieldSize1, userFieldId1 },
    …
    };
```

Defining a field's pSetFunction as NULL indicates that the field is read-only. Similarly, defining a field's pGetFunction as NULL indicates that the field is write-only.

5.4.24.1.1.2 Table Group Field Array

Table groups typically contain multiple rows, holding multiple instances of the same field data. The first field in a table group is the key field, and has the property that it uniquely identifies a specific row in the table, in a way that does not change over time.

The get function for the key field is not needed since get functions for table groups require the key field value as a parameter (see 5.4.24.1.1.3). Similarly, the set function for the key field is not needed either since modifying a set function would change the identity of the row. Instead of get/set functions, the DDM specifies *add* and *delete* row functions that manipulate rows in a table group.

The key field declaration structure described in Table 5-73 differs from other field declarations since it specifies add and delete functions instead of get and set functions. The DDM specifies the remaining fields in a table group using the basic field declaration structure.

Table 5-73 Key Field Declaration Structure

Member	Type	Description
pAddFunction	I2O_PARAMS_FIELD_ACCESS_FUNC	Pointer to function to add row to group.
pDelFunction	I2O_PARAMS_FIELD_ACCESS_FUNC	Pointer to function to delete row from group.
fieldSize	I2O_COUNT	Key field size in bytes
userFieldId	U32	Arbitrary constant associated with field.

The IRTOS passes the key field's userFieldId value as an argument to the add or delete function. The DDM might use this value to identify the particular table.

A typical table group definition might be:

```
I2O_PARAMS_FIELD_DEF tableGroup [] =
    {
      { pAddFunction, pDelFunction, fieldSize0, userFieldId0 },
      { pSetFunction1, pGetFunction1, fieldSize1, userFieldId1 },
      { pSetFunction2, pGetFunction2, fieldSize2, userFieldId2 },
    …
    };
```

Defining pAddFunction as NULL indicates that the group does not support addition of rows via the configuration interface. Similarly, defining pDelFunction as NULL indicates that the group does not support deletion of rows via the configuration interface.

5.4.24.1.1.3 Field Get/Set Functions

The pSetFunction and pGetFunction members of a field declaration are pointers to functions with the following definition:

```
U8 setGetFunction (I2O_OBJ_CONTEXT context, U32 userFieldId,

                   I2O_ADDR32 pKey, I2O_ADDR32 pValue);
```

where:

`context` is the value the DDM supplied in the **i2oCfgParamMsgReply()** or **i2oCfgDialogMsgReply()** `function`. A DDM may use this value to distinguish the specific parameter groups for a particular TID, since DDMs typically create an instance of a group for each of its TIDs.

`pKey` is the address of a buffer containing a key value. For set/get operations on fields within table groups, the key value identifies the row on which an operation is carried out. For scalar groups the get and set functions must ignore the `pKey` parameter.

`pValue` is the address of a buffer containing or receiving the value of the field, and

userFieldId is the userFieldId value supplied in the field declaration,

The DDM can implement these functions in any way it sees fit. For example, the DDM may obtain field values from data structures, compute them from other data, or obtain them from hardware devices. Note that field declarations contain no type information; the get and set functions must know a priori the appropriate type defined for a specific field.

Note that status is not returned via the standard IRTOS status pointer and error handling mechanism because these are not IRTOS API functions. Instead, the DDM’s function returns an 8-bit status value directly.

Table 5-74 Field Get/Set Function Return Values

Value	Description
I2O_PARAMS_STATUS_SUCCESS	Operation completed successfully.
I2O_PARAMS_STATUS_SCALAR_ERROR	Operation on a field in a scalar group failed for an unspecified reason.
I2O_PARAMS_STATUS_TABLE_ERROR	Operation on a field in a table group failed for an unspecified reason.
I2O_PARAMS_STATUS_BAD_KEY_ABORT	Operation on a field in a table group failed due to a unrecognized key value being supplied. The IRTOS should abort the current operation.
I2O_PARAMS_STATUS_BAD_KEY_CONTINUE	Operation on a field in a table group failed due to a unrecognized key value being supplied. The IRTOS may continue the current operation with further key values if desired.

5.4.24.1.1.4 Table Row Add/Delete Functions

The pAddFunction and pDelFunction members of a key field declaration are pointers to functions with the following definition:

```
U8 addDelFunction (I2O_OBJ_CONTEXT context, U32 userFieldId,
                   I2O_ADDR32 pKey);
```

where:

context is the value the DDM supplies in the **i2oCfgParamMsgReply()** or **i2oCfgDialogMsgReply()** call.

pKey is the address of a key value, and

userFieldId is the value supplied in the key field declaration,

The add function creates a new row with the specified key field. The DDM initializes all other fields to reasonable default values. The delete function removes the row with the specified key from the table.

Table 5-75 Row Add/Delete Function Return Values

Value	Description
I2O_PARAMS_STATUS_SUCCESS	Operation completed successfully.
I2O_PARAMS_STATUS_TABLE_ERROR	Operation failed for an unspecified reason.
I2O_PARAMS_STATUS_BAD_KEY_ABORT	Operation failed due to a bad key value being supplied. The IRTOS should abort the current operation.
I2O_PARAMS_STATUS_BAD_KEY_CONTINUE	Operation failed due to a bad key value being supplied. The IRTOS may continue the current operation with further key values if desired.

An add function should return a *BAD_KEY* status if the key value already exists. A delete function should return a *BAD_KEY* status if the key value does not exist.

5.4.24.1.2 GetKeys Function

For scalar groups the pGetKeysFunction member of a group structure must be NULL. For table groups the pGetKeysFunction member must not be NULL . This is a pointer to the function that provides the IRTOS with a list of key field values from rows currently in the group. The function's design allows the IRTOS to make repeated calls, each obtaining successive subsets of the keys in the group. The function has the following definition:

```
U8 getKeysFunction (I2O_OBJ_CONTEXT context, U16 groupNum,
                    I2O_ADDR32 pPrevKey, I2O_COUNT resultBufLen,
                    I2O_ADDR32 pResultBuf, U16 * pNumKeysReturned,
                    U16 * pNumKeysTotal);
```

where:

context is the value the DDM supplies in the **i2oCfgParamMsgReply()** or **i2oCfgDialogMsgReply()** call. Since DDMs typically create an instance of a group for each of its TIDs, a DDM may use this value to distinguish the specific TID whose parameter groups are being accessed.

groupNum indicates the group from which the function obtains the key values.

pNumKeysTotal is the address where the function writes how many keys are currently present in the group, that is, how many rows are in the group.

pNumKeysReturned is the address where the function writes how many key values have been written into the result buffer.

pPrevKey is a pointer to a key value previously obtained via this function. If pPrevKey is NULL, the function returns key values starting at the first row in the group. Otherwise it should return keys starting from the row immediately after the row having the key specified by pPrevKey. In the case that pPrevKey indicates the last row in the group, the function should return successfully with pNumKeysReturned set to zero.

pResultBuf is a pointer to a buffer into which the function writes successive key values.

resultBufLen is the size of pResultBuf buffer. Key values should be written into pResultBuf until it is filled.

Table 5-76 GetKeysFunction Return Values

Value	Description
I2O_PARAMS_STATUS_SUCCESS	Operation completed successfully.
I2O_PARAMS_STATUS_TABLE_ERROR	Operation failed for an unspecified reason.
I2O_PARAMS_STATUS_BAD_KEY_ABORT	Operation failed due to an unrecognized key value being supplied at `pPrevKey`. The current operation should be aborted.
I2O_PARAMS_STATUS_BAD_KEY_CONTINUE	Operation failed due to an unrecognized key value being supplied at `pPrevKey`. The current operation may be continued with further key values if desired.

5.4.24.1.3 Table Clear Function

The pClearFunction member of a group declaration structure for scalar groups must be NULL. For table groups the clear function is optional. If the table does not support the clear function the pClearFunction value is set to NULL.. The clear function deletes all rows from the table. The function has the following definition:

```
U8 clearFunction (I2O_OBJ_CONTEXT context, U16 groupNum);
```

where:

`context` is the value the DDM supplies in the **i2oCfgParamMsgReply()** or **i2oCfgDialogMsgReply()** call. Since DDMs typically create multiple instances of a group for each of its TIDs, a DDM may use this value to distinguish the specific TID whose table is being cleared.

`groupNum` indicates the group to be cleared.

Table 5-77 GetKeysFunction Return Values

Value	Description
I2O_PARAMS_STATUS_SUCCESS	Operation completed successfully.
I2O_PARAMS_STATUS_TABLE_ERROR	Operation failed for an unspecified reason.

5.4.24.2 Event Notification Support

Both the **i2oCfgParamMsgReply()** and **i2oCfgDialogMsgReply()** functions provide support for automation of FIELD_MODIFIED event handling (see Chapter 6, ***UtilEventRegister*** message). The &eventClients parameter provides the IRTOS with a list of clients that registered for event notification. Should the IRTOS successfully change a parameter group field during the course of message processing, the IRTOS issues ***UtilEventRegister*** reply messages to each client in the list who registered for the FIELD_MODIFIED event. If this support is not desired, the DDM sets &eventClients to NULL.

The event client list pointed to by &eventClients has the following format:

Table 5-78 Event Client List Structure

Member	Type	Description
clientCount	I2O_COUNT	Number of clients in client array
reserved	U32	Reserved
clientList	I2O_EVENT_CLIENT_INFO[]	Array of client information structures

The client information structure, I2O_EVENT_CLIENT_INFO has the following format:

Table 5-79 Event Client Info Structure

Member	Type	Description
TargetAddress	I2O_TID	Local DDM/Device TID
Initiator Address	I2O_TID	Event client TID
InitiatorContext	U64	Client initiator context. If 32-bit context fields are being used, the initiator context should appear in the least significant four bytes of this field.
Flags	U32	Flags: Bit 0: reserved Bit 1: A 0 indicates 32-bit context field sizes, in which case the most significant four bytes of the context fields below are ignored. A 1 indicates 64-bit context field sizes. Bits 2-31: reserved
TransactionContext	U64	Client transaction context. If 32-bit context fields are being used, the transaction context should appear in the least significant four bytes of this field.
EventMask	U32	Client event mask

6
Class Specifications

Each I/O class has a message-based interface designated by one of the message class specifications. For each class, this includes messages and a protocol for replying to them. This chapter defines each class and its messages.

6.1 General Requirements

Messages fall into three categories:

- utility messages common to all classes
- base class messages specific to one class
- private messages - not part of this specification

Every device must support utility messages and the base messages for its registered class. Section 6.1.3 defines the utility messages. Some utility messages contain fields that are class-specific. Sections 6.2 through 6.10.7 provide that class-specific detail and define the base messages for each class.

Section 6.1.2 provides generic requirements for replies. Each class defines additional detail based on the reply templates in Chapter 3. For reply details specific to each class, refer to sections 6.2 through 6.10.

Unless otherwise specified, a DDM and its registered devices must support all utility and base class messages for their registered class.

6.1.1 Class Codes

Each device is registered as a particular class and is assigned a TID. The class identifier (i.e., ClassID) identifies the class of messages that may be sent to that TID. The ClassID contains an OrganizationID, a Version identifier, and a 12-bit ClassCode, as shown in the figure below. Classes defined by this specification use the OrganizationID of 0000h and Version 1h.

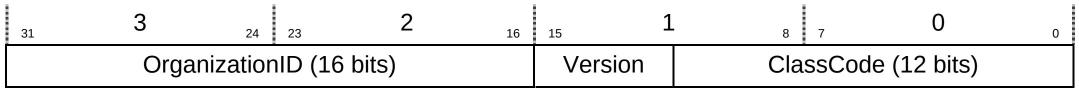


Figure 6-1. Structure for ClassID

6.1.1.1 I$_2$O Standard Class Codes

Table 6-1 gives the ClassCode assignments.

Table 6-1. Class Code Assignments

ClassCode	Class Name (***I2O_CLASS***_xxx)	Description
000h	*_EXECUTIVE*	Manages I/O platforms (e.g., system initialization, system configuration, and peer-to-peer connection)
001h	*_DDM*	Manages the device drivers
010h	*_RANDOM_BLOCK_STORAGE*	Abstraction of a block storage device, such as a hard disk drive or CD ROM drive.
011h	*_SEQUENTIAL_STORAGE*	Abstraction of a sequential storage device, such as a tape drive
020h	*_LAN*	Abstraction of a local area network port, such as an Ethernet or Token Ring controller
030h	*_WAN*	Abstraction of a wide area network port, such as an ATM controller
040h	*_FIBRE_CHANNEL_PORT*	Abstraction of a Fibre Channel port, manages the port itself (reserved)
041h	*_FIBRE_CHANNEL_PERIPHERAL*	Abstraction of a Fibre Channel connection, manages a set of sessions between two fibre channel devices (reserved)
051h	*_SCSI_PERIPHERAL*	Abstraction of a SCSI device
060h	*_ATE_PORT*	Manages the ATE controller
061h	*_ATE_PERIPHERAL*	Abstraction of an ATE device
070h	*_FLOPPY_CONTROLLER*	reserved - Manages the floppy disk controller
071h	*_FLOPPY_DEVICE*	reserved - Abstraction of a floppy disk device
080h	*_BUS_ADAPTER_PORT*	Manages an adapter's port to a secondary bus
090h-09Fh	*_PEER_PEER*	Reserved for Peer-to-peer components

6.1.1.2 Private Message Classes

This specification supports private message classes. A private message class may be defined by any member of the I$_2$O SIG and allows new technology to take advantage of the I$_2$O architecture in a proprietary manner. A message class is a formal interface describing the messages that can be sent to the interface and the replies that will return. A private message class must meet all of the generic requirements specified in chapter 3 and in section 6.1, including utility messages.

The I$_2$O SIG assigns each member organization an OrganizationID which it specifies in the ClassID portion of its private class ID (shown in Figure 6-1). The member organization administers the ClassCode. A DDM that supports that class simply registers one or more devices with that private Class ID.

6.1.1.3 Sub Class Information

Unless specified otherwise, the SubClassInfo field in the Logical Configuration Table shall be the value returned in Parameter Group 0000h, Field 0.

6.1.2 Replies to Request Messages

Reply messages are identified by the REPLY bit in the message header's MessageFlags field. Replies fall into two categories: failed messages and processed messages. Failed messages cannot be processed. They include messages that cannot be delivered, or contain invalid or missing data. Failed messages pertain only to message processing and do not include transactions that fail due to error conditions within the DDM or its hardware. The reply for a failed message is independent of the message class or its function and is specified in section 6.1.2.1.

Note: If the IOP cannot deliver a request to the DDM, due to a system state change, system reconfiguration, or suspended DDM, the IOP returns that message via the ***FaultNotification*** reply, per section 6.1.2.1. For this case, the IOP sets the FAIL bit in the MessageFlags field. The DDM never sets the FAIL bit in the MessageFlags field. If the DDM receives a request with an unknown Function code or an ill-formed message, it replies with a ***Transaction Error Reply Message*** as specified in Chapter 3. Otherwise, the DDM indicates the generic message status in the ReqStatus field and provides a detailed status code in the DetailedStatusCode field.

The term *normal reply* denotes a reply to a request that the target successfully processed (i.e. not a Transaction Error). Normal replies do not have the FAIL bit set in the MessageFlags field. A normal reply depends on both the message class and function. To standardize the normal reply format across classes, this specification defines two default reply templates:

1. for single transactions -- a single reply acknowledging a single transaction
2. for multiple transactions -- a single reply acknowledging multiple transactions from multiple messages

Each class specification defines by Function whether a DDM replies with a single or multiple transaction reply message.

Chapter 3 provides the reply templates for single and multiple transaction messages. All reply structures are based on these default templates. Generally, each class defines a default reply based on these templates. The definition of each request Function either specifies a unique reply payload structure, or indicates that the default structure is used.

The Detailed Status Codes for the Executive Class, DDM Class, Utility Class and Transaction Error replies are specified in Chapter 3. Otherwise each message class defines its own class specific codes.

6.1.2.1 Message Failure Reply

Messages fail when the message layer cannot deliver the message. A DDM's inability to perform the request is an error, not a message failure.

Two mechanisms convey a message's failure to its sender. When a request cannot be delivered, the message layer returns a generic ***FaultNotification*** reply to the initiator of the failed request message, as specified in Chapter 3. When a reply cannot be delivered, there is no mechanism to reply to a failed reply, so the failing module creates and sends an ***UtilReplyFaultNotify*** request message (as specified in section 6.1.3.14) to the sender.

6.1.2.2 Normal Replies

Chapter 3 provides the templates for a normal reply. A normal reply is a message with the REPLY bit set and the FAIL bit reset (see MessageFlags field in Chapter 3). The class definitions in this chapter specify message functions, the reply template, and class-specific fields for the reply.

6.1.3 Utility Messages

Request messages are divided into two major groups: utility messages and base messages. Utility messages are common across class drivers, and their support is mandatory. This section defines the utility messages. Function code assignments for utility messages are specified in Table 6-2.

Table 6-2. Utility Message Function Codes

Function Name	Description
UtilNOP	Do nothing
UtilAbort	Abort previous transaction(s)
UtilClaim	Request use of the device
UtilClaimRelease	Release claim
UtilConfigDialog	Perform a configuration dialogue
UtilDeviceRelease	Release ownership of a device
UtilDeviceReserve	Acquire ownership of a device
UtilEventAck	Acknowledge an event
UtilEventRegister	Turn on/off event notification
UtilLock	Request temporary exclusive control of device
UtilLockRelease	Release lock
UtilParamsGet	Read device parameters
UtilParamsSet	Set device parameters
UtilReplyFaultNotify	Notify module of transport failure of a reply message it generated

All utility messages are single-transaction messages. Typically, the MessageFlags field for requests should be set to 00h (for 32-bit context size) or 02h (for 64-bit context size). For a normal reply, the MessageFlags field should contain C0h (for 32-bit context size) or C2h (for 64-bit context size). Since some requests provide an SGL, the value of the VersionOffset field depends on the location of the SGL. Since all utility replies are single transaction replies, the VersionOffset field should be set to 01h for all replies.

6.1.3.1 Abort Message

The ***UtilAbort*** function causes the device to abort all transactions from the initiator with a Function matching FunctionToAbort and TransactionContext matching TransactionContextToAbort.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilAbort	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
FunctionToAbort	AbortType	reserved		16 (24)
TransactionContextToAbort				20 (28)

offset values for 64-bit context field size in ()

Figure 6-2. *UtilAbort* Request Message

Fields

AbortType

Value	Description
00h	*EXACT_ABORT* - Abort any transactions from this initiator that have a Function code matching the FunctionToAbort and a TransactionContext matching Transaction ContextToAbort.
01h	*FUNCTION_ABORT* - Abort any transactions from this initiator that have a function code matching the FunctionToAbort (any TransactionContext).
02h	*TRANSACTION_ABORT* - Abort any transactions from this initiator that have a TransactionContext matching TransactionContextToAbort (any Function code).
03h	*WILD_ABORT* - Abort all transactions from this initiator (any function, and any Transaction Context).
04h	*CLEAN_EXACT_ABORT* - Abort only messages from this initiator that have a Function code matching the FunctionToAbort, and a TransactionContext matching TransactionContextToAbort that have not started any data transfer (Clean Abort).
05h	*CLEAN_FUNCTION_ABORT* - Abort any messages from this initiator that have a Function code matching the FunctionToAbort (any TransactionContext) that have not started any data transfer (Clean Abort).
06h	*CLEAN_TRANSACTION_ABORT* - Abort any messages from this initiator that have a TransactionContext matching TransactionContextToAbort (any Function code) that have not started any data transfer (Clean Abort).
07h	*CLEAN_WILD_ABORT* - Abort all messages from this initiator (any Function, and TransactionContext) that have not started any data transfer (Clean Abort)

FunctionToAbort Value to match with message Function field.

TransactionContextToAbort Value to match with TransactionContext field. The field is the same as the size as TransactionContext fields.

The reply contains the count of transactions aborted (which may be zero). The target must reply to all aborted transactions before replying to the ***UtilAbort*** request. The ***UtilAbort*** reply always indicates success and has the FINAL bit set. If no matches are found, the CountOfAbortedMessages is set to zero. The Clean Abort command aborts messages in the input queue and does not affect transactions being executed.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilAbort	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
CountOfAbortedMessages				16 (24)

offset values for 64-bit context field size in ()

Figure 6-3. *UtilAbort* Reply Message

The target must reply to each aborted transaction. This may be a batch reply, if the class definition so allows. The reply status code of *STATUS_CODE_ ABORT_NO_DATA_TRANSFER* and *STATUS_CODE_ ABORT_PARTIAL_TRANSFER* signifies that the abort succeeded and that the buffers will not be accessed again for that transaction. In this case, the FINAL bit is set to 1. If the DDM attempts an abort, but cannot ascertain its status (e.g., the hardware does not respond) or cannot guarantee that the buffers will not be accessed, then the DDM returns a *STATUS_CODE_ABORT_DIRTY* status to the respective transaction and the FINAL bit is not set. The CountOfAborted messages includes all transactions returned with a status of *STATUS_CODE_ABORT_NO_DATA_TRANSFER*, *STATUS_CODE_ ABORT_PARTIAL_TRANSFER*, or *STATUS_CODE_ABORT_DIRTY*.

By definition, all transactions aborted by a Clean Abort have a reply status of *STATUS_CODE_ABORT_NO_DATA_TRANSFER*. Note that Clean Abort is not the exact opposite of Dirty Abort. Rather, Clean Abort cancels a transaction only if no data has transferred, while Dirty Abort means that the conclusion of the abort is not known due to some failure.

6.1.3.2 Claim Message

The ***UtilClaim*** request notifies the target that the initiator asks to use its base class functions. The normal reply to this request is a default single-transaction reply with no ReplyPayload. A successful reply indicates that the target will accept base class messages from the initiator.

The potential user identifies the type of control it requires as follows:

Primary User: In the simple model an OSM is the primary user, and in the stacked driver model an ISM is the primary user. The primary user establishes executive control over the device and thus there can only be one primary user. Three other user classifications allow for multiple hosts (clustered operation), peer-to-peer, and management.

Management User: Generally management agents do not claim devices unless they desire to change parameters. A conflict arises if both a management agent and the primary user set the same parameter. Each class definition identifies which parameters may be modified by a management user. Unless specifically stated, the management user does not modify parameters. In addition to managed objects, there are certain base class messages that also cause a conflict with the primary user. To allow flexibility, this specification does not provide specific requirements for base class messages and allows each message class to specify its own requirements. Note that many class definitions are in their infancy and do not particularly address active management.

Authorized User: The primary user may wish to authorize additional users allowing them to send base class messages and modify parameters. The extent of control that an authorized user possesses is privately defined by the primary user. How the primary user determines authorized users is private and outside the scope of this specification. The device being claimed assumes that an authorized user has the same rights as the primary user. Authorized users are affirmed by the primary user. The authorized user category gives the primary user explicit control over which devices can access the services of the claimed device. The authorized user category is useful for clustered environments, redundant hosts, and for peer-to-peer.

Secondary User: The primary user may not wish to expressly authorize each additional user and therefore enables the claimed device for secondary service. Unless otherwise prohibited, a secondary user may send any base class messages but does not modify operational parameters. Operation of secondary users should be transparent to the primary service. The secondary user category is useful for peer-to-peer operation.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilClaim	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
ClaimType	reserved	ClaimFlags		16 (24)

offset values for 64-bit context field size in ()

Figure 6-4. ***UtilClaim*** **Request Message**

Fields

ClaimFlags Each bit of this field represents a different claim attribute. The only bits defined are:

Bit 0 = Exclusive – reserved. Bits 4 and 5 replace this bit definition.

Bit 1 = ResetSensitive – setting this bit indicates that the initiator needs to synchronize with any resets to the device. For such a reset, the target rejects any further base class requests from the initiator until an appropriate ***UtilEventAck*** message is received. As an example, when a SCSI bus reset is performed, all SCSI peripheral devices on that bus are reset, and each device generates a *DEVICE_RESET* event to its users and rejects further messages until the reset is acknowledged.

Bit 2 = StateSensitive – setting this bit indicates that the initiator needs to synchronize with any state changes of the device (such as media or volume change of a storage device). If such a change occurs, the device rejects any further base class requests from the initiator until an appropriate ***UtilEventAck*** message is received.

Bit 3 = CapacitySensitive – setting this bit indicates that the initiator needs to synchronize with any capacity changes of the device. If such a change occurs, the device rejects any further base class requests from the initiator until it receives an appropriate ***UtilEventAck*** message.

Bit 4 Peer Service Class Disabled: The target rejects ***UtilClaim*** requests from secondary users when this bit is set. If a secondary user already claimed the device, then the target rejects this request. This bit is only meaningful in the ***UtilClaim*** request from the primary user. If target rejects the request because a secondary user already exists, the initiator might consider the ***ExecDeviceRelease*** message.

Bit 5 Management Service Class Disabled: The target rejects subsequent ***UtilClaim*** messages from management users when this bit is set. If a management user has already claimed the device, then the target rejects this request. This bit is only meaningful in the ***UtilClaim*** request from the primary user. If the target rejects the request because a management user already exists, the initiator might consider the ***ExecDeviceRelease*** message.

ClaimType A value that represents the intent of the claiming entity:

PRIMARY_USER: A device may have only one primary user. If a ***UtilClaim*** request indicates a primary user and another user already claimed this device as a primary user, then the target rejects the request. If it accepts the request, it rejects subsequent ***UtilClaim*** requests indicating a primary user, until the primary user sends a ***UtilClaimRelease*** message.

AUTHORIZED_USER: The primary user authorizes alternate users using the ***UtilParamsSet*** utility message (Group F006h - Authorized User Table). When a received ***UtilClaim*** request indicates an authorized user, the target verifies that the initiator is authorized. If not, the target rejects the request.

SECONDARY_USER: The target rejects a ***UtilClaim*** request from a secondary user if the primary user disabled peer service class.

MANAGEMENT_USER: The target rejects a ***UtilClaim*** request from a management user if the primary user disabled management service class.

Note

Because all OSMs use the same TID (001h), a DDM cannot differentiate between OSMs. Once a DDM accepts a **UtilClaim** *request from a primary user, it rejects all* **UtilClaim** *requests indicating a primary user, even if they are from the same TID as the first. The OS is responsible for any correlation between OSMs that need to share a resource.*

6.1.3.3 Claim Release Message

The ***UtilClaimRelease*** function notifies the target that the initiator releases its claim. The normal reply to this request is a single transaction reply with no ReplyPayload.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilClaimRelease	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
ClaimType	reserved	ReleaseFlags		16 (24)

offset values for 64-bit context field size in ()

Figure 6-5. *UtilClaimRelease* Request Message

Fields

ClaimType A value that identifies the user type being released (see ***UtilClaim*** request).

ReleaseFlags Each bit of this field represents a different release attribute. The only bit defined is:

Bit 0 = Conditional – May only be set by the primary user. When this bit is set, the DDM retains the Authorized User List and rejects a primary claim unless it is from a TID specified in the list. If the primary user does not set this bit in its release, then the target clears the authorized list and may reject base class messages from any authorized user. When the primary user releases the claim, the target assumes that all authorized users are aware.

Note

Because all OSMs use the same TID (001h), a DDM cannot differentiate between OSMs. A DDM accepts a **UtilClaimRelease** *without verification. An OSM or ISM must not send a* **UtilClaimRelease** *unless it sent the* **UtilClaim.**

6.1.3.4 Configuration Dialogue Message

The ***UtilConfigDialog*** request in Figure 6-6 starts or continues a configuration session with the target. The SGL identifies the buffer where the target deposits its configuration template and supplies any data from the operator. The reply to this request is a single transaction reply with no ReplyPayload.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
UtilConfigDialog	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
PageNumber				16 (24)
SGL				20 (28)

offset values for 64-bit context field size in ()

Figure 6-6. *UtilConfigDialog* Request Message

Fields

PageNumber The host side service extracts the PageNumber from the HTML GET. Page 0 is the device's home page and all other page numbers result from links in the pages displayed by the device.

SGL The SGL specifies one or two buffers. The first buffer contains the target's HTML text for the next screen. The second buffer, when present, contains the HTML results from the previous display passed from the browser to the target (the form data from an HTML POST). That text is in the form field1=value1&field2=value2 and usually represents new values for selected fields in selected groups.

VersionOffset A value of 51h for 32-bit context size, and 71h for 64-bit context size.

The normal reply to an ***UtilDialogGet*** message is a default reply with no ReplyPayload and the ReqStatus field set to *STATUS_CODE_SUCCESS*. The reply buffer contains HTML text.

If an error occurs while processing the message, the ReqStatus field is set to *STATUS_CODE_ERROR_NO_DATA_TRANSFER* or *STATUS_CODE_ERROR_PARTIAL_TRANSFER* and the DetailedStatusCode field adds information about the nature of the error (see detailed status codes in Chapter 3). In some cases, the HTML text in the reply buffer provides error information.

6.1.3.5 Device Release Message

The ***UtilDeviceRelease*** message releases the exclusive ownership of a device acquired via the ***UtilDeviceReserve*** message. Only the primary user or an authorized user should send this message.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilDeviceRelease	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)

offset values for 64 bit context field size in ()

Figure 6-7. *UtilDeviceRelease* Request Message

6.1.3.6 Device Reserve Message

The ***UtilDeviceReserve*** message directs the target to acquire exclusive ownership of a device using the appropriate low level reservation protocol. This command is appropriate for devices that have multiple paths (multiple DDMs, IOPs, or units controlling the same device).

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilDeviceReserve	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)

offset values for 64 bit context field size in ()

Figure 6-8. *UtilDeviceReserve* Request Message

When this command is successful, then all users (see ***UtilClaim*** message) have access to the device via base class messages. If the request fails, then no users have access. The following table depicts the normal reply codes the target generates.

Table 6-3. Device Reserve Reply Codes

ReqStatus	DetailedStatusCode	Meaning
_SUCCESS	_SUCCESS	This driver acquired the device.
_SUCCESS	*_UNSUPPORTED_FUNCTION*	No low level reservation protocol defined for this class or sub-class (such as for a LAN port). This driver acquired the device by default.
_ERROR_NO_DATA_TRANSFER	*_DEVICE__NOT_AVAILABLE*	Another driver already acquired the device.
_ERROR_NO_DATA_TRANSFER	*_UNSUPPORTED_FUNCTION*	Device does not support the low level reserve protocol.
_ERROR_NO_DATA_TRANSFER	*_INAPPROPRIATE_FUNCTION*	Not a valid request for this message class (such as for Exec or DDM class).

For an ISM, the low level reservation protocol is:

1. The ISM issues a ***UtilDeviceReserve*** message to each device claimed by the target of the original request.
2. The reply to the original request reflects the aggregate of the replies to the secondary requests.

6.1.3.7 Event Acknowledge Message

This request acknowledges an event and enables normal operation. This message is necessary only for events that cause the device to synchronize with the user (see ***UtilClaim*** request message ClaimFlags).

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilEventAck	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
EventIndicator				16 (24)
EventData :				20 (28)

offset values for 64-bit context field size in ()

Figure 6-9. *UtilEventAck* Request Message

Fields

EventIndicator The EventIndicator field from the ***UtilEventRegister*** reply message.

EventData The EventData field from the ***UtilEventRegister*** reply.

The reply includes the information from the request as follows:

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilEventAck	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
EventIndicator				16 (20)
EventData :				20 (28)

offset values for 64-bit context field size in ()

Figure 6-10. *UtilEventAck* Reply Message

6.1.3.8 Event Registration Message

This request turns event notification on and off. A variety of event categories exist, and this message provides independent control of each category. ***UtilEventRegister*** enables and disables events generic to all message classes as well as events specific to each class. The target does not reply to this message until one of the enabled events occurs. The target generates an event reply message for each enabled event (one reply per event instance per registered event) until the client masks off that event. Initially, all events are masked until the client sends an ***UtilEventRegister*** message.

31 **3** 24	23 **2** 16	15 **1** 8	7 **0** 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilEventRegister	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
EventMask				16 (24)

offset values for 64-bit context field size in ()

Figure 6-11. *UtilEventRegister* Request Message

Fields

EventMask Each bit of this field represents a different category of events. If the bit is set, the event reporting for that category is enabled. If the value is zero, event reporting for that category is disabled. Bits 0–15 are defined by the specific message class. The remaining bits are defined in Table 6-4.

31 **3** 24	23 **2** 16	15 **1** 8	7 **0** 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilEventRegister	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
EventIndicator				16 (24)
EventData :				20 (28)

offset values for 64-bit context field size in ()

Figure 6-12. *UtilEventRegister* Reply Message

Fields

EventIndicator A 32-bit enumerated value specifying the source of the event category that triggered this event. Only one bit can be set, and its location corresponds to the event category that triggered this event as described by Table 6-4.

EventData Information about the event. The structure of this field depends on the event category specified by the EventIndicator, as described in Table 6-5 or by the specific message class.

The target uses the InitiatorAddress plus the InitiatorContext and TransactionContext of ***UtilEventRegister*** requests to correlate requests. This allows multiple OSMs to each register an ***UtilEventRegister*** with the same device. A single OSM must use the same InitiatorContext and TransactionContext for all ***UtilEventRegister*** requests.

Table 6-4. EventIndicator Assignments

Bit	Event Name (*I2O_EVENT_IND*_xxx)	Description
31	*_STATE_CHANGE*	Reports changes in the driver's state, which alters its capability to perform I/O transactions.
30	*_GENERAL_WARNING*	Reports warnings that eventually may cause the driver to change state.
29	*_CONFIGURATION_FLAG*	Configuration requested.
28	*_ LOCK_RELEASE*	A lock that caused requests to be rejected has been released. It is not necessary to send a message if no requests had been rejected.
27	*_CAPABILITY_CHANGE*	One or more capabilities has changed (e.g., Event Capability).
26	*_DEVICE_RESET*	A device reset has occurred. Assume any programmed state altered.
25	*_EVENT_MASK_MODIFIED*	Causes a reply to this message (an acknowledgment).
24	*_FIELD_MODIFIED*	Notifies users when a field is changed by a UtilParamsSet request.
23	*_VENDOR_EVENT*	Provides a dedicated event for vendor specific purposes.
22	*_DEVICE_STATE*	Reports generic device state changes.

Table 6-5. EventData for Generic Events

Event Name	EventData
StateChange	An 8-bit value as follows: 00h = Returned to normal operation 01h = Suspended/quiesced 02h = Reset/restarted 03h = I/O device/service not available – recovery expected 04h = I/O device/service not operational – recovery not expected 05h = Quiesce request received 10h = Failed – recovery expected 11h = Faulted – recovery not expected
GeneralWarning	An 8-bit value as follows: 00h = Returned to normal operation 01h = Error threshold exceeded 02h = Media fault detected
ConfigFlag	No data
LockRelease	No data
CapabilityChange	A bit map as follows: Bit 0 = Category other than the following Bit 1 = Event capability changed
DeviceReset	No data
EventMaskMod	New event mask
FieldMod	GroupNumber, FieldIdx, and KeyValue (if applicable) of the modified field.
VendorUnique	Content is specified by the user. Size of the event data up to the remainder of the message frame.
DeviceState	An 32-bit value as follows: bit 0 Sensor state change

6.1.3.9 Lock Message

The ***UtilLock*** function notifies the target that the initiator temporarily requests exclusive use of its base class functions. The normal reply to this request is a single transaction reply with no ReplyPayload. A successful reply indicates that the initiator has exclusive rights until it sends a ***UtilLockRelease*** message. The target either holds or rejects messages received by other initiators until it receives the ***UtilLockRelease*** message. Also see the ***DeviceReserve*** message.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilLock	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)

offset values for 64-bit context field size in ()

Figure 6-13. *UtilLock* Request Message

Note

Because all OSMs use the same TID (001h), a DDM cannot differentiate between OSMs. Once a DDM accepts an **UtilLock***, it accepts all messages from the reserving TID. The OS is responsible for any correlation and locking between OSMs that need to share a resource.*

I$_2$O provides the ***UtilLock*** and ***UtilLockRelease*** messages for supporting devices with multiple paths and users.

Note

Because these concepts are relatively new, many aspects of locking devices are purposely omitted in this version of the document. These include how long a resource can remain locked, when a device is locked, how long requests by other users are held before they are rejected, and how to recover from a dead initiator.

OSMs and ISMs must only lock a resource for the minimum necessary time. To prevent deadlock, the target uses the reply error code *DEVICE_LOCKED* to reject requests from other users. The target should not reject messages unless the reserve and/or deferred request has exceeded a reasonable time limit. The target also provides a *LockRelease* event to notify users that the lock is released.

6.1.3.10 LockRelease Message

The ***UtilLockRelease*** function notifies the target that the initiator is releasing its reservation. The normal reply to this request is a single transaction reply with no ReplyPayload.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
UtilLockRelease	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)

offset values for 64-bit context field size in ()

Figure 6-14. *UtilLockRelease* Request Message

Note

Because all OSMs use the same TID (001h), a DDM cannot differentiate between OSMs. A DDM accepts a **UtillockRelease** *without verification. An OSM or ISM must not send a* **UtilLockRelease** *unless it successfully sent the* **UtilLock**.

6.1.3.11 NOP Message

The ***UtilNOP*** function contains no payload and does not solicit a reply. TargetAddress and InitiatorContext should be set to zero. MessageFlags should be set to *Request, no_Fail*. Upon receiving a ***UtilNOP***message, the IOP returns the message frame to the free list. A module that receives a ***UtilNOP*** message releases the message and does not reply.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize=3		MessageFlags	VersionOffset = 01h	0
UtilNOP	InitiatorAddress	TargetAddress		4
InitiatorContext				8

Figure 6-15. *UtilNOP* Request Message

6.1.3.12 ParamsGet Message

The ***UtilParamsGet*** message allows parameter values to be retrieved from device parameter groups. The request consists of one or more query operations.

The ***UtilParamsGet*** request message contains a SGL specifying one or two buffers. The first buffer contains read operations. The second buffer is a reply buffer where the target places the results of those operations. If the SGL contains only one buffer, the target returns the results in the reply payload.

Parameters are grouped in sets identified by a GroupNumber. Groups of generic parameters are defined for all device classes and each message class defines additional parameter groups. The initiator provides an *operation list* identifying which parameters of which groups the target will return. The result is a list of those parameter values. The ***UtilParamsGet*** request message is shown in Figure 6-16. The normal reply to the ***UtilParamsGet*** request is a single transaction reply with no ReplyPayload if a reply buffer was specified. Otherwise, the reply is a single transaction reply with the results listed in the ReplyPayload. The latter provides a low overhead mechanism for reading a few parameter values.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
UtilParamsGet	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
OperationFlags				16 (24)
SGL :				20 (28)

offset values for 64-bit context field size in ()

Figure 6-16. *UtilParamsGet* Request

Field

OperationFlags Reserved

SGL Provides one or two buffers. The first buffer contains the operation list (see Chapter 3) that identifies which parameters are to be returned. If a second buffer is present, the target places the results in it as described in Chapter 3. If the second buffer is not present, then the target places results in the ReplyPayload.

VersionOffset A value of 51h for 32-bit context size and 71h for 64-bit context size.

In the absence of errors, the target sets the reply's ReqStatus field to *STATUS_CODE_*SUCCESS, and the reply buffer (or ReplyPayload) contains the *result list* in the format described in Chapter 3.

Some error conditions prevent the result information from being returned. Under such conditions, the target sets the reply's ReqStatus field to *STATUS_CODE_ERROR_NO_DATA_TRANSFER*. For errors that allow partial completion, the target sets the reply's ReqStatus field to *STATUS_CODE_ERROR_PARTIAL_TRANSFER*. In either case the DetailedStatusCode field provides additional information about the error. (See Chapter 3).

In addition to the status codes in the reply message, read operations return error information identifying the source of the error.

6.1.3.13 ParamsSet Message

See ***UtilParamsGet*** message for more details. The ***UtilParamsSet*** message modifies the value of parameters. The SGL of the ***UtilParamsSet*** request specifies one or two buffers. The first buffer contains modify operations specifying the fields and their respective new values. The second buffer is a reply buffer where the target places the results of those operations. If the SGL contains only one buffer, the target returns the results in the ReplyPayload.

The initiator provides an *operation list* identifying which parameters of which groups the target will modify along with the new values. The result is a status list. The ***UtilParamsSet*** request message is shown in Figure 6-17. The normal reply to the ***UtilParamsSet*** request is a single transaction reply with no ReplyPayload if a reply buffer was specified. Otherwise, the reply is a single transaction reply with the results list in the ReplyPayload. The latter provides a low-overhead mechanism for modifying a few parameter values.

The rules for error reports are the same as for the ***UtilParamsGet*** message. See Chapter 3 for DetailedStatusCodes.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
UtilParamsSet	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
OperationFlags				16 (24)
SGL				20 (28)

offset values for 64-bit context field size in ()

Figure 6-17. *UtilParamsSet* Request Message

Fields

OperationFlags Reserved

SGL Provides one or two buffers. The first buffer contains the operation list (see Chapter 3) that identifies the parameters to modify. If a second buffer is present, the target places the result in it as described in Chapter 3. If the second buffer is not present, then the target places the results in the ReplyPayload.

VersionOffset A value of 51h for 32-bit context size and 71h for 64-bit context size.

6.1.3.14 Reply Message Failure Notification

This message is generated by the transport layer when a reply message cannot be delivered. This message is never sent in response to a reply message with the FAIL bit set in the MessageFlags field. There is no reply to this message.

<table>
<tr><th>31 3 24</th><th>23 2 16</th><th>15 1 8</th><th>7 0 0</th><th></th></tr>
<tr><td colspan="2">MessageSize</td><td>MessageFlags</td><td>VersionOffset = 01h</td><td>0</td></tr>
<tr><td><b><i>UtilReplyFaultNotify</i></b></td><td>InitiatorAddress</td><td colspan="2">TargetAddress</td><td>4</td></tr>
<tr><td colspan="4">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="4">TransactionContext (null)</td><td>12 (16)</td></tr>
<tr><td>FailureCode</td><td>Severity</td><td>HighestVersion</td><td>LowestVersion</td><td>16 (24)</td></tr>
<tr><td colspan="2">FailingHostUnitID</td><td>reserved</td><td>FailingIOP_ID</td><td>20 (28)</td></tr>
<tr><td colspan="4">AgeLimit</td><td>24 (32)</td></tr>
<tr><td colspan="4">Original Message MFA (Low 32 bits)</td><td>28 (36)</td></tr>
<tr><td colspan="4">Original Message MFA (High 32 bits)</td><td>32 (38)</td></tr>
</table>

offset values for 64-bit context field size in ()

Figure 6-18. *UtilReplyFaultNotify* Request Message

The message payload is identical to the payload described in the Fault Reply Message Structure (see Chapter 3 and 6.1.2.1 Message Failure Reply). The differences are:

- This is a request message.
- The FAIL bit in the MessageFlags field is not set.
- InitiatorContext is any arbitrary value.

6.2 Executive Class

Executive class messages are defined in Chapter 4, *I$_2$O Shell Interface Specification*.

6.3 Device Driver Class

Device driver class messages are defined in Chapter 5, *I$_2$O Core Specification*.

6.4 Random Block Storage Class

6.4.1 Overview

A block storage device provides random access to a permanent storage medium. The DDM registers a different BSA class device for each logical drive it provides. The client, typically an

OSM, performs block storage operations by sending requests to, and listening for replies from, a BSA class device.

This section describes the I_2O block storage abstraction model for block-oriented storage devices. Storage devices in this class include: standard disk drives, CD-ROM drives, WORM drives, and removable media devices, including devices managed by juke boxes. Tape devices and variable block size devices have their own class definition separate from the block storage class. The I_2O Block Storage Abstraction (BSA) layer is the primary interface host operating systems use to access block storage devices.

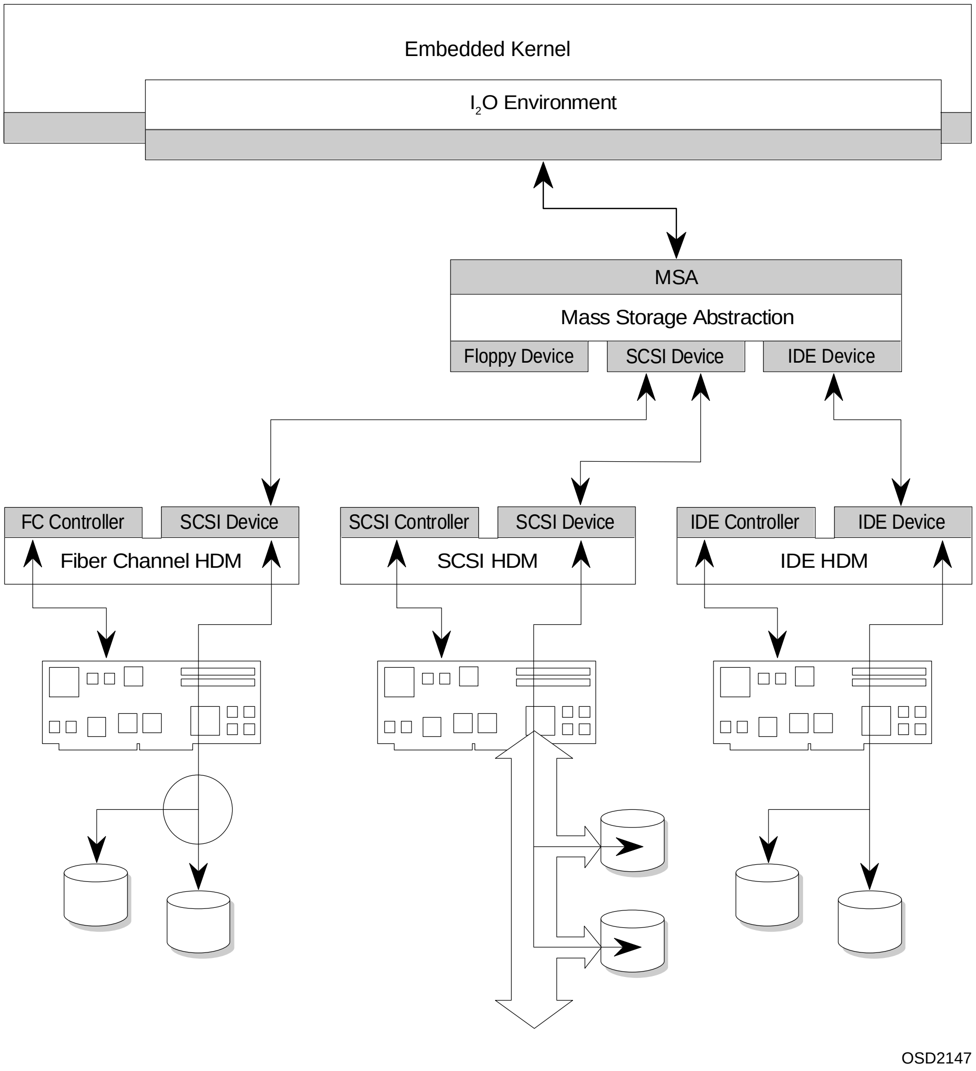


Figure 6-19. Block Storage Abstraction

6.4.2 Operational Model

The abstraction layer employs a request/reply model on top of the I_2O messaging interface. The requester builds a request and sends it to the IOP, where the abstraction layer runs. In a monolithic DDM, the request is handled internally and the reply is sent to the requester. This might happen with a SCSI device DDM that exposes not only a SCSI device class but also a block storage device class. A stacked block storage class driver builds a new message targeted at the device’s actual class driver, for example, the SCSI device class for SCSI devices. The new message is sent to the lower-level class driver. A reply is eventually received from the lower-level class driver, which replies to the original request, thus concluding the transaction.

The following sections describe basic operating principles for block storage class DDMs and its client (typically an OSM). In general, the term *device* applies to the abstract interface produced by the message class and DDM refers to the software providing that interface. The client sends messages to that device by indicating the TID for that device in the message’s TargetAddress field. The terms *physical device* and *drive* refer to the physical device or facility the DDM controls. *Behavior* of the device includes the software as well as its physical device(s).

6.4.2.1 Transaction Ordering

Transaction ordering is the responsibility of the host. A DDM need only preserve request ordering to the level specified in the OrderedRequestDepth parameter from *OPERATIONAL_CTL* group. This parameter indicates the number of requests for which a DDM can maintain ordering. If the OrderedRequestDepth is two and the client sends three requests, ordering is not guaranteed. The client may try to change this parameter, but the DDM may not honor the request if it cannot support its depth. The host determines the actual ordering depth supported by performing a Get after setting this parameter.

If the client issues overlapping requests beyond the OrderedRequestDepth, they are not guaranteed to occur in the order they arrived.

6.4.2.2 Clearing Error Conditions

In general, when a client receives an operation error, other than a timeout, it assumes that the DDM attempted to clear the error condition and retried the request. (See Table 6-17, *OPERATIONAL_CONTROL*, regarding retry counts.)

Besides retrying the request, the client can choose another method of recovery. As part of the reply message, the DDM returns a detailed status code that supplies the failed status. The client should try to recover through the interfaces provided by the block storage abstraction layer (i.e. the Block Storage class). It should not bypass that layer by dealing with the underlying interfaces (e.g., SCSI Device or SCSI Controller classes).

For example, if the physical storage device is a SCSI drive, additional information can be obtained via the TID of the corresponding SCSI device or the TID of its SCSI adapter.

Resetting the device: In general, the following behavior should be observed.

- A DDM does not, on its own, perform a reset that causes a loss of data to other devices, unless those devices are not operating.
- When the DDM detects an error condition, it attempts to clear the condition and complete the request in a non-destructive manner.

  Example: In the case of a SCSI Check Condition, the DDM performs the request-sense sequence to clear the condition. If successful, a warning is reported back to the host as a successful operation with retries.

- The last status information from an error condition is retained by the DDM for logging purposes (see section 6.4.7 Get/Set Parameters group = **Error_Log**)
- Issues with device/bus specific behavior (e.g., freezing queues in SCSI) should be transparent to the client. That is, the DDM must clear the condition and unfreeze the queue.

6.4.2.3 Timing Out Requests

Many block storage requests have some form of associated timeout. Timeout values are accessible via a ***UtilParamsSet*** (GROUP=**Operational_Control**) request. Each request function has some form of associated multiplier or formula to determine the timeout value, as stated in Table 6-6. Some functions provide a TimeMultiplier in the message that multiplies the timeout for a particular request. A TimeMultiplier of zero suspends the timeout for that transaction.

Table 6-6. Timeout Formula

Function	Timeout Formula
BsaBlockRead	TimeMultiplier x (RWVTimeoutBase + (RWVTimeout x size/64k))
BsaBlockReassign	TimeMultiplier x timeout_Base
BsaBlockWrite	TimeMultiplier x (RWVTimeoutBase + (RWVTimeout x size/64k))
BsaBlockWriteVerify	TimeMultiplier x (RWVTimeoutBase + (RWVTimeout x size/64k))
BsaCacheFlush	TimeMultiplier x timeout_Base
BsaDeviceReset	TimeMultiplier x TimeoutBase
BsaMediaEject	no specified timeout
BsaMediaFormat	no specified timeout
BsaMediaLock	no specified timeout
BsaMediaMount	no specified timeout
BsaMediaUnlock	no specified timeout
BsaMediaVerify	TimeMultiplier x (RWVTimeoutBase + (RWVTimeout x size/64k))
BsaPowerMgt	TimeMultiplier x PowerdownTimeout
BsaStatusCheck	TimeoutBase uSec
Utility messages	TimeoutBase uSec

If TimeMultiplier is zero, then do not timeout.

6.4.2.4 Reset Requests

When the host requests a reset from a block storage device, the following behavior should be observed:

- A reset of a block storage device, either hard or soft, must not cause data loss to any other device.
  Example: In the case of SCSI-based devices, this means a reset of a device (i.e., SCSI device reset) does not cause a SCSI bus reset, unless the reset avoids data loss for other devices (e.g., tapes with in-progress requests).
- A soft reset of a block storage device clears initiator requests.

- A hard reset of a block storage device resets the device. All queued and outstanding requests return to the initiator with a device reset error code. All initiators that need to re-establish non-default operating parameters should indicate that in their ***UtilClaim*** request and request event notification with the DeviceReset event enabled. When a hard reset occurs, the DDM replies to the event notification request with the DeviceReset indicator set. For initiators that register for ResetSensitivity, the DDM rejects I/O requests from that initiator until an ***UtilEventAck*** request is received. For initiators that do not register for ResetSensitivity, the DDM continues servicing requests immediately after the device is on-line.
- The host must escalate the reset. That is, if a hard reset to the mass storage TID does not clear the condition, the host should reset the next device in the hierarchy. The host must locate and reset the port or bus associated with the device. The host learns the hierarchy by the ***UtilParamsGet*** messages (see section 6.4.7) and examines the logical configuration table structures. A reset message sent to a TID for a bus controller clears the condition in a hardware-specific manner.

  Example: A soft reset of a SCSI adapter resets the SCSI bus or its interface. A DDM does not issue a soft SCSI reset unless it will avoid data loss to another device.

  A hard reset of a SCSI adapter resets the card itself. A DDM does not issue a hard SCSI reset unless it can guarantee that doing so will avoid data loss to another device.

6.4.2.5 Event Sensitivity

In the ***UtilClaim*** request, the client indicates its sensitivity to particular events. This mechanism protects the client against device resets, media mounts and dismounts. That is, in-flight requests -- messages sent after an event, but before the client acknowledges it -- must be rejected until state can be reestablished.

When an event occurs to which the host is sensitive, the DDM rejects all requests, except for utility requests, until an ***UtilEventAck*** is received. This lets the host reestablish state before it enables the DDM to accept requests.

6.4.2.6 Managing Device State

All operational control for managing the state of media is performed through either the Get/Set operations or the power management message.

An example initialization sequence from the host for a mass storage device is:

1. Obtain logical configuration table data
2. ***Identify***
3. ***UtilEventRegister*** to enable event posting
4. ***UtilClaim*** to specify event sensitivity
5. Determine the state of the device from Get operations
6. Power up the device

6.4.2.7 Changing Hardware Specific Settings

The configuration dialogue changes hardware specific parameters, such as a SCSI configuration (SCSI ID). The client is not involved in this type of activity.

Examples of configuration dialogue for SCSI include:

```
SetSyncData

       SyncOffset  – offset in bytes

       SyncPeriod  – Period of Synchronous, granularity is 4 ns

SetInitiatorID

       U8   InitiatorID  – New initiator ID

... Response

SetAsync

       Async/Sync flag

SetQueuing

       Enable/Disable – for SCSI this is Tagging

SetDisconnect

       Don’tDisconnect
```

6.4.2.8 Devices With Special Capabilities

The Block Storage class model accommodates devices that support multiple hardware access paths and dynamic capacity changes.

6.4.2.8.1 Devices Supporting Multiple Hardware Access Paths

Some devices, especially sophisticated storage subsystems, provide multiple hardware access paths to the device for redundancy, load balancing, or sharing. The client determines this through a ***UtilParamsGet*** request. The host OS must take the appropriate action (see ***UtilDeviceReserve*** message).

6.4.2.8.2 Devices Capable of Dynamic Capacity Changes

The DDM registers capacity changes via event notification. This allows seamless, dynamic additions to storage devices in the host. Dynamic capacity changes are not yet fully supported. You can obtain this capability by combining configuration dialogue and capacity event notification.

6.4.2.9 Device Statistics

The request to get or set operating parameters (***UtilParamsGet*** or ***UtilParamsSet***) defines groups for controlling and gathering statistical information. Statistical data is not yet defined but will conform to industry standards.

6.4.2.10 Block Storage Hierarchy

Through the ***UtilParamsGet*** request, a client learns the hierarchy of devices as well as the logical-to-physical relationships. The DDM provides logical-to-physical information for storage subsystem management and does not suggest bypassing the logical device or an abstracted interface in favor of a physical device or interface.

6.4.2.11 Initialization Hierarchy of a Block Storage Device

When block storage devices come online, the I_2O system can initialize them in various ways.

Example 1: host configuration case

1. The DDM for the SCSI port presents the device as a SCSI peripheral with no user.
2. The IOP updates its logical configuration table and sends an ***ExecLctNotify*** reply to the host.
3. The host assigns the SCSI device to a mass storage DDM via the ***ExecDeviceAssign*** message. The mass storage driver may be the same DDM that controls the SCSI port, may be provided by an independent driver, or may be unspecified. If the ***ExecDeviceAssign*** message specifies permanent assignment, the next time the system initializes, the IOP follows example 2; otherwise, it repeats example 1.
4. The IOP assigns the SCSI device to the mass storage DDM via the ***DdmDeviceAttach*** message. The mass storage driver may be the same DDM that controls the SCSI port or may be provided by an independent driver. If the device is attached to the same DDM, then next time the system initializes, the DDM may follow example 3.
5. The mass storage driver claims the device via the ***UtilClaim*** message.
6. The SCSI peripheral driver sets the userTID in the logical configuration table to the TID of the mass storage DDM.
7. The mass storage driver presents the device as a mass storage device with no user.
8. The IOP updates its logical configuration table and sends an ***ExecLctNotify*** reply to the host.
9. The host mass storage OSM claims the mass storage device via the ***UtilClaim*** message.
10. The mass storage driver sets the mass storage device’s userTID in the logical configuration table to the TID of the client.

Example 2: IOP configuration case

1. The DDM presents the device as a SCSI peripheral.
2. The IOP updates its logical configuration table and assigns the device to a mass storage device driver via the ***DdmDeviceAttach*** message. The mass storage driver may be the same DDM that controls the SCSI port, or an independent driver. If the device is attached to the same DDM, then next time the system initializes, the DDM may follow example 3.
3. The mass storage driver claims the device via the ***UtilClaim*** message.
4. The SCSI peripheral driver sets the userTID in the logical configuration table to the TID of the mass storage DDM.
5. The mass storage driver presents the device as a mass storage device.
6. The IOP updates its logical configuration table and sends an ***ExecLctNotify*** reply to the host.

7. The host mass storage OSM claims the mass storage device via the ***UtilClaim*** message.
8. The mass storage driver sets the mass storage device’s userTID in the logical configuration table to the TID of the client.

Example 3: immediate configuration case

1. When the device comes on line, the SCSI/mass storage driver immediately presents it as a mass storage device with no user. In this instance, the DDM presents a SCSI peripheral device with its own TID as the userTID.
2. The IOP updates its logical configuration table and sends an ***ExecLctNotify*** reply to the host.
3. If the DDM receives an ***DdmDeviceRelease*** message for the SCSI peripheral, it is recommended that the next time the system initializes, the DDM responds as in example 1 or 2.
4. The host mass storage OSM claims the mass storage device via the ***UtilClaim*** message.
5. The mass storage driver sets the mass storage device’s userTID in the logical configuration table to the TID of the client.

6.4.2.12 Service and Management

Setting the configuration dialogue bit causes an ***ExecLctNotify***, initiating a configuration dialogue. The configuration dialogue manages device configuration or reconfiguration. Three examples are:

- Initial device configuration
- Replacing failed drives in RAID subsystems.
- Managing maintenance conditions, such as fan failure.

6.4.3 Device Addressing

The addressing of devices is based on the I_2O addressing model, where all devices in the system have an I_2O address (TID) associated with them; that includes disks, CD-ROMs, and tape drives. Physical interface-specific addressing (e.g., SCSI target IDs or LUNs) is excluded from the model.

The absence of the physical addressing elements reduces the number of special cases within the message formats and allows new physical interfaces to plug into the model without inventing a new class.

6.4.4 Block Storage Reply Messages

A reply is generated for every request. The reply structure for block storage requests is the Default Reply Template for Single Transaction Requests defined in Chapter 3 (see section 6.1.2.2).

Note: The DDM never sets the FAIL bit in the MessageFlags field. If the DDM receives a request with an unknown Function code or an ill-formed message, it replies with a ***Transaction Error Reply Message*** as specified in Chapter 3.

When a DDM aborts a request because the system state changes, it sends a final reply for each outstanding transaction as an error (not an abort).

6.4.4.1 Block Storage Status Codes

Table 6-7. DetailedStatusCode for Block Storage Operations

DetailedStatusCode	Description
BSA_SUCCESS	Success – no warnings
BSA_ACCESS_ERROR	Protocol retry. A bus protocol error was successfully retried. Retry count supplied. GET_LAST_LOGGING_DATA can retrieve hardware specific status.
BSA_ACCESS_VIOLATION	The device is locked for exclusive access by another party.
BSA_BUS_FAILURE	The operation failed due to a problem with the operation of the bus.
BSA_DEVICE_FAILURE	Device does not respond or responds with fault
BSA_DEVICE_NOT_READY	Device is not ready for access. Device state may be determined by ***UtilParamsGet*** group = DEVICE_INFO and/or group = POWER_CTL.
BSA_DEVICE_RESET	After a reset, all requests aside from utility messages (base class requests) are returned with *DEVICERESET* status until the reset is acknowledged by the user.
BSA_MEDIA_ERROR	Media retry. Device was forced to retry to read/write the data. Retry count supplied. GET_LAST_LOGGING_DATA can retrieve hardware specific status.
BSA_MEDIA_FAILURE	The operation failed due to an error on the medium
BSA_MEDIA_LOCKED	Media locked by another party.
BSA_MEDIA_NOT_PRESENT	Removable medium not loaded.
BSA_PROTOCOL_FAILURE	The operation failed due to a communication problem with the device.
BSA_TIMEOUT	The operation failed because the time-out value specified for this request has been exceeded.
BSA_VOLUME_CHANGED	After a volume change, all requests aside from utility messages (base class requests) are returned with *VOLUMECHANGED* status until the event is acknowledged.
BSA_WRITE_PROTECTED	The medium is write protected or read only.

DetailedStatusCodes values do not apply equally to transfers, errors, and aborts. The DDM must report an appropriate code.

6.4.4.2 Successful Completion

For requests that completed without error, the DDM sets the ReqStatus to reflect successful completion, and the ReplyPayload indicates the total bytes transferred.

<table>
<tr><th>31 3 24</th><th>23 2 16</th><th colspan="8">15 1 8</th><th>7 0 0</th><th></th></tr>
<tr><td colspan="2">MessageSize</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>x</td><td>0</td><td>VersionOffset = 01h</td><td>0</td></tr>
<tr><td>Function</td><td colspan="5">InitiatorAddress</td><td colspan="5">TargetAddress</td><td>4</td></tr>
<tr><td colspan="11">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="11">TransactionContext</td><td>12 (16)</td></tr>
<tr><td><i>_SUCCESS</i></td><td>RetryCount</td><td colspan="9">DetailedStatusCode</td><td>16 (24)</td></tr>
<tr><td colspan="11">TransferCount</td><td>20 (28)</td></tr>
</table>

offset values for 64-bit context field size in ()

Figure 6-20. Successful Completion Reply Message for Block Storage Class

Fields

RetryCount — Number of retries to complete the request. A value of zero means success on the first try. Reason for failures provided in the DetailedStatusCode.

DetailedStatusCode — If non-zero, contains a warning code describing the nature of recovered operation. See Table 6-7.

TransferCount — The number of bytes transferred. For transactions not involving data transfer this value is set to zero.

6.4.4.3 Aborted Operation

Transactions aborted at the request of the originator have the ReqStatus set to *STATUS_CODE_ABORT_NO_DATA_TRANSFER*, *STATUS_CODE_ABORT_PARTIAL_TRANSFER*, or *STATUS_CODE_ABORT_DIRTY* and there is no ReplyPayload.

<table>
<tr><th>31 3 24</th><th>23 2 16</th><th colspan="8">15 1 8</th><th>7 0 0</th><th></th></tr>
<tr><td colspan="2">MessageSize</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>x</td><td>0</td><td>VersionOffset = 01h</td><td>0</td></tr>
<tr><td>Function</td><td colspan="5">InitiatorAddress</td><td colspan="5">TargetAddress</td><td>4</td></tr>
<tr><td colspan="11">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="11">TransactionContext</td><td>12 (16)</td></tr>
<tr><td>AbortCode</td><td>RetryCount</td><td colspan="9">DetailedStatusCode (Table 6-7)</td><td>16 (24)</td></tr>
</table>

offset values for 64-bit context field size in ()

Figure 6-21. Aborted Operation Reply Message for Block Storage Class

AbortCode — Contains a generic status code as specified in chapter 3 (*STATUS_CODE_ABORT_NO_DATA_TRANSFER*, *STATUS_CODE_ABORT_PARTIAL_TRANSFER* or *STATUS_CODE_ABORT_DIRTY*).

6.4.4.4 Progress Reports

Progress replies address block storage requests that can take a relatively long time to complete, such as a verify operation of an entire device. Progress replies are only appropriate for certain requests as indicated by the ProgressReport bit in each request's ControlFlags field. When enabled, the DDM sends periodic progress reports, as shown in Figure 6-22. The ReqStatus value of *STATUS_CODE_PROGRESS_REPORT* identifies a progress report. Progress replies are always followed by a reply with a final ReqStatus (e.g., *STATUS_CODE_SUCCESS*, *STATUS_CODE_ABORT_NO_DATA_TRANSFER*, *STATUS_CODE_ABORT_PARTIAL_TRANSFER*, *STATUS_CODE_ERROR_NO_DATA_TRANSFER*, *STATUS_CODE_ERROR_PARTIAL_TRANSFER*, *STATUS_CODE_ABORT_DIRTY_ERROR_DIRTY*). The exception is when the progress report is returned in response to a ***BsaStatusCheck*** request. In this case, the reply is a final message.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		1 0 0 0 0 0 x 0	VersionOffset = 01h	0
Function	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
_Progress_Report	RetryCount	DetailedStatusCode (Table 6-7)		16 (24)
Reserved			PercentComplete	20 (28)

offset values for 64-bit context field size in ()

Figure 6-22. Progress Report Reply Message for Block Storage Class

Fields

PercentComplete The percentage complete value, 0-100

Progress messages are in 1% resolution. The DDM determines the rate at which these messages are sent, although a rate of one progress message per second is a good guideline.

A ***StatusCheck*** message is an alternative to enabling progress reporting (see section 6.4.6.14).

6.4.4.5 Error Reports

For requests that do not complete successfully, the Reply's ReqStatus indicates *STATUS_CODE_ERROR_NO_DATA_TRANSFER*, *STATUS_CODE_ERROR_PARTIAL_TRANSFER*, or *STATUS_CODE_ERROR_DIRTY,* the DetailedStatusCode contains a detailed error code, and location information is returned in the AdditionalInformation (see chapter 3) portion of the reply message, as shown in Figure 6-23.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		1 1 0 0 0 0 x 0	VersionOffset=01h	0
Function	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
ReqStatus	RetryCount	DetailedErrorCode (Table 6-7)		16 (24)
TransferCount				20 (28)
LogicalByteAddress				24 (32)
(64 bits)				28 (36)

offset values for 64-bit context field size in ()

Figure 6-23. Unsuccessful Completion Reply Message for Block Storage Class

Fields

ReqStatus — Contains generic status codes as specified in chapter 3 (*STATUS_CODE_ERROR_NO_DATA_TRANSFER*, *STATUS_CODE_ERROR_PARTIAL_TRANSFER*, or *STATUS_CODE_ERROR_DIRTY*).

DetailedErrorCode — Contains as specific a code as possible to describe the nature of the failure See Table 6-7.

LogicalByteAddress — Points to the location of the error. For a transaction not associated with a logical byte address (such as device reset), the DDM must set this value to zero.

If the LogicalByteAddress in the reply differs from the starting address in the request, the initiator should not assume successful I/O operation to any blocks in between. The content of any block involved in a write operation that did not complete successfully is unknown. Similarly, any memory blocks involved in a read operation that did not complete are unknown. The recommended recovery policy should be to retry the operation for all blocks and not a subset.

6.4.5 Support for Utility Messages

6.4.5.1 Lock

The ***UtilLock*** request causes the DDM to guarantee the initiator exclusive access to the device until a ***UtilLockRelease*** request is issued by the same initiator. See the ***UtilLock*** message in section 6.1.3.14.

6.4.5.2 Lock Release

A ***UtilLockRelease*** request cancels a previous reservation. Issuing a ***UtilLockRelease*** request without completing a ***UtilLock*** causes the DDM to fail the request.

6.4.5.3 Events

Each Block Storage class device supports generic events specified in section 6.1.3.4 and the ***UtilEventRegister*** request. In addition, each device supports the following block storage events.

Table 6-8. BSA EventIndicator Assignments

Event Name	Bit	Description
VolumeLoad	0	New medium has been loaded onto the device
VolumeUnload	1	The medium on the device has been unloaded
VolumeUnloadRequest	2	An external unload request and medium is locked. The client must unlock the medium before the unload request can be honored.
CapacityChange	3	The capacity of the device has changed.
SCSI_SMART	4	Reports SCSI SMART data is received

Table 6-9. EventData for Block Storage Events

Event Name	EventData
VolumeLoad	No data
VolumeUnload	No data
VolumeUnloadRequest	No data
CapacityChange	No data
SCSI_SMART	SCSI ASC and ASCQ (2 bytes)

6.4.5.4 Getting and Setting Parameters

Both the client (service user) and management use the ***UtilParamsGet*** and ***UtilParamsSet*** utility messages specified in 6.1.3, *Utility Messages*, to read and modify parameters for block storage devices. The list of parameter groups for Block Storage class devices is specified in 6.4.7.

6.4.6 Block Storage Request Messages

Table 6-10 lists requests a client can make to a Block Storage class device. The following sections define these requests in alphabetic order.

Table 6-10. Block Storage Request Messages

Function	Description
BsaBlockRead	Read from device to memory
BsaBlockReassign	Reassign block addresses
BsaBlockWrite	Write to the device from memory
BsaBlockWriteVerify	Write to the device from memory then verify
BsaCacheFlush	Write dirty cache to media
BsaDeviceReset	Reset the device
BsaMediaEject	Eject a removable medium from drive mechanism
BsaMediaFormat	Not defined
BsaMediaLock	Prevent media removal
BsaMediaMount	Load a removable medium into drive mechanism
BsaMediaUnlock	Allow media removal
BsaMediaVerify	Verify accessibility of data
BsaPowerMgt	Power management
BsaStatusCheck	Check device status

All Random Block Storage class messages are single-transaction messages. Typically, the MessageFlags field for requests should be set to 00h (for 32-bit context size) or 02h (for 64-bit context size). For a normal completion reply message, the MessageFlags field should contain C0h (for 32-bit context size) or C2h (for 64-bit context size). Since some requests provide an SGL, the value of the VersionOffset field depends on the location of the SGL. Since all replies are single transaction, the VersionOffset field should be set to 01h for all.

6.4.6.1 Block Read

The ***Read_Block*** function transfers bytes from a storage device to system memory.

<table>
<tr><th colspan="2">31 3 24</th><th colspan="2">23 2 16</th><th colspan="2">15 1 8</th><th colspan="2">7 0 0</th><th></th></tr>
<tr><td colspan="4">MessageSize</td><td colspan="2">MessageFlags</td><td colspan="2">VersionOffset</td><td>0</td></tr>
<tr><td colspan="2"><b><i>BsaBlockRead</i></b></td><td colspan="3">InitiatorAddress</td><td colspan="3">TargetAddress</td><td>4</td></tr>
<tr><td colspan="8">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="8">TransactionContext</td><td>12 (16)</td></tr>
<tr><td colspan="2">FetchAhead</td><td colspan="2">TimeMultiplier</td><td colspan="4">ControlFlags</td><td>16 (24)</td></tr>
<tr><td colspan="8">TransferByteCount</td><td>20 (28)</td></tr>
<tr><td colspan="8" rowspan="2">LogicalByteAddress
(64 bits)</td><td>24 (32)</td></tr>
<tr><td>28 (36)</td></tr>
<tr><td colspan="8" rowspan="2">SGL</td><td>32 (40)</td></tr>
<tr><td>n</td></tr>
</table>

offset values for 64-bit context field size in ()

Figure 6-24. *BsaBlockRead* Request Message

Fields:

ControlFlags Options flags specifying the DDM actions for this read (see Table 6-11).

FetchAhead Additional amount of data to be prefetched and cached by the DDM, if possible. Granularity is in one kilobyte increments.

LogicalByteAddress Medium logical address where the operation begins.

SGL Indicates the buffer where the data will be placed.

TargetAddress Address (TID) of the storage device.

TimeMultiplier Multiplier used with RWVTimeoutBase and RWVTimeout (see section 6.4.2.3) to determine request timeout. Zero indicates the timeout is suspended for this request.

TransferByteCount Number of bytes transferred.

VersionOffset A value of 81h for 32-bit context size and A1h for 64-bit context size.

Although the number of bytes to transfer (TransferByteCount) and the logical media address (LogicalByteAddress) have byte granularity, the target device must return an error if either field is not in multiples of the device's logical block size. Typically, this is 512 bytes.

A TransferByteCount of zero is not an error for ***Read_Block*** requests; it simply implies a seek operation with no need for data transfer. In this case, the SGL might not exist and should be considered undefined by the DDM and any DMA support hardware.

Table 6-11. Block Read Control Flags

ControlFlags	Description
Bit 0	Do not retry the request if it fails.
Bit 1	Solo – the request should not be placed into a hardware queue but handled as an individual request.
Bit 2	Cache read – the data read should be cached.
Bit 3	Read with prefetch – the prefetch field is valid and the *n* 1-kilobyte blocks following the end of the request should be read into the DDM's cache.
Bit 4	Cache data – the data read should not be transferred, it should be cached by the DDM. No SGL is provided. The DDM can treat this as an NOP.

6.4.6.2 Block Reassign

The ***BsaBlockReassign*** request remaps the specified regions on the medium. The DDM must not issue a completion status reply until the reassignment is complete.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
BsaBlockReassign	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	reserved		16 (24)
SGL				20 (28)

offset values for 64-bit context field size in ()

Figure 6-25. *BsaBlockReassign* Request Message

VersionOffset A value of 51h for 32-bit context size and 71h for 64-bit context size.

The scatter-gather list describes a set of address/length pairs to reassign. ByteCount must be in multiples of block size.

31 3 24	23 2 16	15 1 8	7 0 0	
ByteCount				0
LogicalByteAddress (64 bits)				4

6.4.6.3 Block Write

The ***Write_Block*** function transfers bytes from system memory to a storage device.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
BsaBlockWrite	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	ControlFlags		16 (24)
TransferByteCount				20 (28)
LogicalByteAddress				24 (32)
(64 bits)				28 (36)
SGL				32 (40)
				n

offset values for 64-bit context field size in ()

Figure 6-26. *BsaBlockWrite* Request Message

Fields:

ControlFlags	Options flags specifying the DDM actions for this write (see Table 6-12).
LogicalByteAddress	Media logical address at which the operation begins.
SGL	The buffer that holds the source data.
TargetAddress	Address of the storage device.
TimeMultiplier	Multiplier used with RWVTimeoutBase and RWVTimeout to determine request timeout (see 6.4.2.3). Zero indicates the timeout is suspended for this request.
TransferByteCount	Number of bytes to transfer.
VersionOffset	A value of 81h for 32-bit context size and A1h for 64-bit context size.

Although the number of bytes to transfer (TransferByteCount) and the logical medium address (LogicalByteAddress) have byte granularity, the target device must return an error if either field is not in multiples of the device's logical block size. Typically this is 512 bytes.

Table 6-12. Block Write Control Flags

ControlFlags	Description
bit 0	Do not retry the request if it fails.
bit 1	Solo – the request should not be placed into a hardware queue but should be handled as an individual request.
bit 2	Do not cache – the DDM should not cache the write (used on sequential writes).
bit 3	Write through cache – the DDM must make the data durable before the request is completed.
bit 4	Write to cache – the DDM can reply before the request is durable on the medium, assuming the data is cached (good for transient, swap pages).

6.4.6.4 Block Write and Verify

The ***BsaBlockWriteVerify*** message asks the target to write the data to the medium and verify that the data is correctly written. Depending on the actual mass storage interface, the DDM may have to write and read to verify the readability of the data.

<table>
<tr><td colspan="2">31 3 24</td><td colspan="2">23 2 16</td><td colspan="2">15 1 8</td><td colspan="2">7 0 0</td><td></td></tr>
<tr><td colspan="4">MessageSize</td><td colspan="2">MessageFlags</td><td colspan="2">VersionOffset</td><td>0</td></tr>
<tr><td colspan="2"><b><i>BsaBlockWriteVerify</i></b></td><td colspan="3">InitiatorAddress</td><td colspan="3">TargetAddress</td><td>4</td></tr>
<tr><td colspan="8">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="8">TransactionContext</td><td>12 (16)</td></tr>
<tr><td colspan="2">reserved</td><td colspan="2">TimeMultiplier</td><td colspan="4">ControlFlags</td><td>16 (24)</td></tr>
<tr><td colspan="8">TransferByteCount</td><td>20 (28)</td></tr>
<tr><td colspan="8">LogicalByteAddress
(64 bits)</td><td>24 (32)
28 (36)</td></tr>
<tr><td colspan="8">SGL</td><td>32 (40)</td></tr>
</table>

offset values for 64-bit context field size in ()

Figure 6-27. *BsaBlockWriteVerify* Request Message

Fields:

ControlFlags — Options flags specifying the DDM actions for this write (see Table 6-13). Other bits reserved.

VersionOffset — A value of 81h for 32-bit context size and A1h for 64-bit context size.

Table 6-13. Block Write & Verify Control Flags

ControlFlags	Description
bit 0	Do not retry the request if it fails.
bit 1	Solo – the request should not be placed into a hardware queue but handled as an individual request.
bit 2	Do not cache – the DDM should not cache the write (used on sequential writes).

6.4.6.5 Cache Flush

The ***BsaCacheFlush*** request causes the DDM to write all dirty-cached data to the medium. The DDM must not issue a completion status reply until all data is actually on the medium. It is the DDM's responsibility to ensure that any dirty data cached downstream is also flushed to the medium.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
BsaCacheFlush	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	ControlFlags		16 (28)

offset values for 64-bit context field size in ()

Figure 6-28. *BsaCacheFlush* Request Message

Fields:

ControlFlags Specifies options for performing this command.

Bit 7: ProgressReport- When this bit is set, the device posts Progress Reports (see 6.4.4.4). When it is not set, the device does not post Progress Reports.

Other bits reserved.

A failure should be reported only if the medium is no longer accessible (removed and hard failure).

6.4.6.6 Device Reset

The ***BsaDeviceReset*** function brings the device into a known state. The DDM aborts all outstanding requests (i.e., those queued for the device, as well as those in progress) and returns them with an appropriate abort status.

<table>
<tr><th colspan="2">31 3 24</th><th colspan="2">23 2 16</th><th colspan="2">15 1 8</th><th colspan="2">7 0 0</th><th></th></tr>
<tr><td colspan="4">MessageSize</td><td colspan="2">MessageFlags</td><td colspan="2">VersionOffset = 01h</td><td>0</td></tr>
<tr><td colspan="2"><b><i>BsaDeviceReset</i></b></td><td colspan="3">InitiatorAddress</td><td colspan="3">TargetAddress</td><td>4</td></tr>
<tr><td colspan="8">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="8">TransactionContext</td><td>12 (16)</td></tr>
<tr><td colspan="2">reserved</td><td colspan="2">TimeMultiplier</td><td colspan="4">ControlFlags</td><td>16 (24)</td></tr>
</table>

offset values for 64-bit context field size in ()

Figure 6-29. *BsaDeviceReset* Request Message

Fields:

ControlFlags — Options for performing this command.

Bit 0: Hard/Soft - When this bit is set, the DDM performs a hard reset, which causes the device to reinitialize. Otherwise, the DDM completes a soft reset, which clears the protocol to the medium.

Other bits reserved

TimeMultiplier — Multiplier used with TimeoutBase to determine request timeout (see section 6.4.2.3, *Timing Out Requests*). Zero indicates that the timeout is suspended for this request.

6.4.6.7 Media Eject for Removable Media

The ***BsaMediaEject*** request causes the medium in the drive to eject. If the device is locked as a result of a prior ***BsaMediaLock*** request, the device fails any ***BsaMediaEject*** requests until after the device receives a ***BsaMediaUnlock*** request. A failure also results if this device does not support the ejection.

<table>
<tr><th colspan="2">31 3 24</th><th colspan="2">23 2 16</th><th colspan="2">15 1 8</th><th colspan="2">7 0 0</th><th></th></tr>
<tr><td colspan="4">MessageSize</td><td colspan="2">MessageFlags</td><td colspan="2">VersionOffset = 01h</td><td>0</td></tr>
<tr><td colspan="2"><b><i>BsaMediaEject</i></b></td><td colspan="3">InitiatorAddress</td><td colspan="3">TargetAddress</td><td>4</td></tr>
<tr><td colspan="8">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="8">TransactionContext</td><td>12 (16)</td></tr>
<tr><td colspan="8">MediaIdentifier*</td><td>16 (24)</td></tr>
</table>

offset values for 64-bit context field size in ()

Figure 6-30. *BsaMediaEject* Request Message

* Optional parameter, -1 indicates whatever is currently mounted on the drive.

6.4.6.8 Media Format

The ***BsaMediaFormat*** message is not defined in this version of the document. Low-level format is hardware dependent and few devices support it.

6.4.6.9 Media Lock

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
BsaMediaLock	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
MediaIdentifier*				16 (24)

offset values for 64-bit context field size in ()

Figure 6-31. *BsaMediaLock* Request Message

* Optional parameter, -1 indicates whatever is currently mounted on the drive.

6.4.6.10 Media Mount for Removable Media

The ***BsaMediaMount*** request causes the DDM to attempt to load the removable medium identified by the MediaIdentifier. If the device cannot automatically load the medium, it should return an unsupported status. The medium can optionally lock in the device after the load.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
BsaMediaMount	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
MediaIdentifier				16 (24)
Reserved			LoadFlags	20 (28)

offset values for 64-bit context field size in ()

Figure 6-32. *BsaMediaMount* Request Message

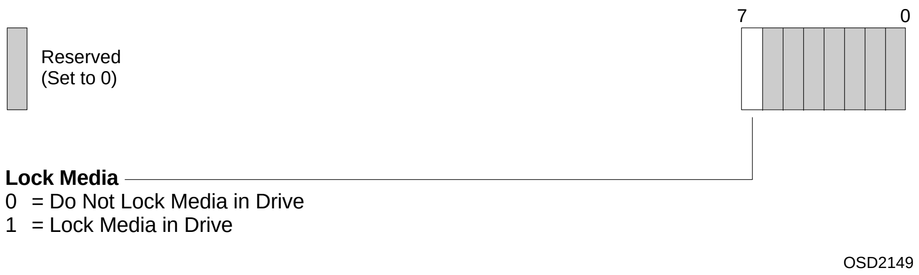


Figure 6-33. Load Flags for *BsaMediaMount*

6.4.6.11 Media Unlock

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
BsaMediaUnlock	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
MediaIdentifier*				16 (24)

offset values for 64-bit context field size in ()

Figure 6-34. *BsaMediaUnlock* Request Message

* Optional parameter, -1 indicates whatever is currently mounted on the drive.

6.4.6.12 Media Verify

The ***BsaMediaVerify*** message asks the target to verify the readability of the blocks on the medium. No data transfer occurs during this operation.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
BsaMediaVerify	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	ControlFlags		16 (32)
ByteCount				20 (28)
LogicalByteAddress				24 (32)
(64 bits)				28, (36)

offset values for 64-bit context field size in ()

Figure 6-35. *BsaMediaVerify* Request Message

Fields:

ControlFlags Options for performing this command.

Bit 6: Correction - When this bit is set, the DDM performs error correction and sparing. When this bit is cleared, the DDM performs no error correction or sparing.

Bit 7: ProgressReport- When this bit is set, the device posts Progress Reports (see section 6.4.4.4). When it is not set, the device does not post Progress Reports.

Other bits reserved

When a progress reply is requested, the DDM is responsible for sending progress messages to the initiator. A final reply with a status of *Success* or *Failure* concludes the request.

The ***BsaMediaVerify*** failure reply contains the failing address and the number of bytes in the blocks that failed. For all blocks between the initial LogicalByteAddress in the request and the LogicalByteAddress in the reply, assume that all status is good. For all blocks past those in the error message, the status is unknown.

6.4.6.13 Power Management

The ***BsaPowerMgt*** command modifies the state of the device.

31 — 3 — 24	23 — 2 — 16	15 — 1 — 8	7 — 0 — 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
BsaPowerMgt	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
Operation	TimeMultiplier	ControlFlags		16 (24)

offset values for 64-bit context field size in ()

Figure 6-36. *BsaPowerMgt* Request Message

Fields:

ControlFlags — Options for performing this command.

Bit 7: ProgressReport- When this bit is set, the device posts Progress Reports (see section 6.4.4.4). When it is not set, the device does not post Progress Reports.

Other bits reserved

TimeMultiplier — Timeout multiplier

Operation — One of the power management operations described in the table below.

Table 6-14. Power Management Operation Values

Operation	Description
01h	Power up partial – power up the device in a minimal state. The volume need not be made available (spun up).
02h	Power up – power the device up completely.
03h	Power up, load – power up the device completely and load medium, if present.
20h	Quiesce device – flush any volatile state out to the volume and quiesce device activity.
21h	Power down partial – power down the device to a minimal state. A volume should be spun down, if present, but not unloaded.
22h	Power down partial, unload – as above, but unload the volume, if removable.
23h	Power down, unload – fully power down the device, unloading the volume, if present.
24h	Power down, retain – fully power down the device, retaining the medium.
other values	reserved

The following table defines the codes used to control the power for SCSI devices.

Table 6-15. Power Management SCSI translation Matrix

Operation	SCSI Operation
0001h	NA
0002h	Start
0003h	Start and load medium
0020h	Stop
0021h	Re-zero
0022h	Stop and eject
0023h	Stop and eject
0024h	Stop
other values	reserved

6.4.6.14 Status Check

Issuing a ***BsaStatusCheck*** request to a device returns either *STATUS_CODE_SUCCESS*, *STATUS_CODE_PROGRESS_REPORT*, or *STATUS_CODE_ERROR0*. *STATUS_CODE_SUCCESS* indicates that the device is online and operating, while *STATUS_CODE_ERROR0* indicates that the device may be operating or not, depending on the detailed error code in the reply. (See Section 6.4.4.) A *STATUS_CODE_PROGRESS_REPORT* reply is returned when the device has a FORMAT or VERIFY operation in progress.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
BsaStatusCheck	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)

offset values for 64-bit context field size in ()

Figure 6-37. *BsaStatusCheck* Request Message

The reply for a ***BsaStatusCheck*** depends on the state of the device. The table below describes replies:

Table 6-16. StatusCheck Replies

ReqStatus (*STATUS_CODE_*xxx)	Description
_ABORT_NO_DATA_TRANSFER	The ***StatusCheck*** request was aborted. The format of this message is an Abort Report (6.4.4.3)
ERROR NO_DATA_TRANSFER	The medium is not available. The reason can be determined by interpreting the detailed status information. The format of this message is an Error Report (6.4.4.5)
_PROGRESS_REPORT	A request is currently active against the device that supports PROGRESS replies (a FORMAT or a VERIFY). The format of this message is a Progress Report (6.4.4.4)
_SUCCESS	The medium is available.

6.4.7 Modifying Configuration and Operating Parameters

Both the client (service user) and management utilize the ***UtilParamsGet*** and ***UtilParamsSet*** utility messages specified in section 6.1.3*, Utility Messages to read and modify parameter for random block storage devices*. The list of parameter groups for the Block Storage class is specified in the following tables.

Table 6-17. BSA Parameters Group

GroupNumber	0000h
GroupType	SCALAR
Name	***DEVICE INFORMATION***
Description	Information to describe a block storage device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	1 byte	DeviceType	Type of storage device. Storage Device Types currently supported: 00 Direct-access read/write 04 Write-once device (WORM) 05 CD-ROM device 07 Optical memory device
1	r	1 byte	NumberOfPaths	Number of access paths to the medium. To support dual/multi-ported devices.
2	r	2 bytes	PowerState	Operation set by the most recent power management message.
3	r	4 bytes	BlockSize	Block size (number of bytes). If medium is removable, report maximum-supported size.
4	r	8 bytes	DeviceCapacity	Device capacity (number of bytes). If medium is removable, report maximum-supported capacity.

GroupNumber	0000h
GroupType	SCALAR
Name	***DEVICE INFORMATION***
Description	Information to describe a block storage device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
5	r	4 bytes	DeviceCapabilitySupport	Device capabilities describes attributes of the device, such as removable, mountable, lockable, etc. Storage Device Capabilities supported: bit 0 Caching bit 1 Multi-path accessible bit 2 Dynamic capacity changes bit 3 Removable (Media) bit 4 Removable (Device) bit 5 Read-only (Security) bit 6 Lockout (Security) bit 7 Boot bypass (Security) bit 8 Compression bit 9 Data Security bit 10 RAID
6	r	4 bytes	DeviceState	Device states bit 0 Caching bit 1 PoweredOn bit 2 Ready bit 3 Media Loaded bit 4 Device Loaded bit 5 Read-only (Security) bit 6 Lockout (Security) bit 7 Boot bypass (Security) bit 8 Compression bit 9 Data Security bit 10 RAID

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0001
GroupType	SCALAR
Name	***OPERATIONAL_CONTROL***
Description	Operational control parameters for a block storage device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r/w	1 byte	AutoReassign	The DDM can auto-reassign blocks.
1	r/w	1 byte	ReassignTolerance	Number of retries before a block is a candidate for reassignment.
2	r/w	1 byte	RetryAttempts	Number of times a DDM/device retries a request before failing.
3	r	1 byte	reserved1	
4	r/w	4 bytes	ReassignSize	Size of the block reassignment (sparing) area. A write to this may fail.
5	r	4 bytes	ExpectedTimeout	Longest time expected for any 64-kilobyte I/O operation (in microseconds). The client uses this value as guidance.
6	r/w	4 bytes	RWVTimeout	Read/Write/Verify timeout increment, per 64-kilobyte block transfer (in microseconds).
7	r/w	4 bytes	RWVTimeoutBase	Base timeout value to add to the timeout increment (in microseconds).
8	r/w	4 bytes	TimeoutBase	Base timeout for standard operations (in microseconds). The timeout value is subject to a multiplier determined by the message type.
9	r/w	4 bytes	OrderedRequestDepth	Indicates the number of requests for which the device can maintain ordering. If the DDM cannot support the requested depth, it sets the value to the maximum supported depth. On a subsequent inquiry, the host obtains the actual depth. This parameter operates on a per-initiator basis. That is, if the host and a peer set the parameter, each has its own OrderedRequestDepth, which the DDM maintains. Most devices/DDMs support a depth of 1. A value of -1 indicates that the device maintains ordering for all requests.
10	r	4 bytes	AtomicWriteSize	Largest size of request (in bytes) that can be atomically written to the medium.

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0002
GroupType	SCALAR
Name	***POWER CONTROL***
Description	Power control parameters for configuring how a block storage device responds during inactivity and recovery from a powered down state.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r/w	4 bytes	PowerdownTimeout	The device powers down if it is not accessed within the allotted time (in microseconds). A zero value indicates the device will never power down.
1	r/w	4 bytes	OnAccess	Determines what the device does when accessed in a powered down state. Any block storage class request against the medium constitutes access. bit 0 PowerUpOnAccess bit 1 LoadOnAccess

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0003
GroupType	SCALAR
Name	***CACHE CONTROL***
Description	Information and control parameters for the cache of a block storage device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 bytes	TotalCacheSize	Total available cache (in bytes).
1	r/w	4 bytes	ReadCacheSize	Total available cache size for reads (in bytes).
2	r/w	4 bytes	WriteCacheSize	Total available cache size for writes (in bytes).
3	r/w	1 byte	WritePolicy	Policy employed by the cache when handling write requests. 00h None/Disabled 01h WriteToCache 02h WriteThruCache
4	r/w	1 byte	ReadPolicy	Policy employed by the cache when handling read requests. 00h None/Disabled 01h ReadCache 02h ReadAheadCache 03h ReadReadAheadCache
5	r/w	1 byte	ErrorCorrection	Error correction scheme. 00h None/Disabled 01h Unknown 02h Other 03h Parity 04h SingleBitECC 05h MultiBitECC

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0004
GroupType	SCALAR
Name	***MEDIA INFORMATION***
Description	

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	Capacity	Formatted capacity (in bytes) for current medium.
1	r	4 bytes	BlockSize	Block size (in bytes) for current medium.

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0005h
GroupType	TABLE
Name	***ERROR_LOG***
Description	Table of information for each error encountered. The OSM uses this information for its system log.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 bytes	ErrorDataIndex	Unique index that identifies each error log entry.
1	r	1 byte	Function	The function code of the request that failed.
2	r	1 byte	RetryCount	The number of times the function was unsuccessfully tried.
3	r	2 bytes	DetailedErrorCode	Detailed error code describing the error or failure.
4	r	2 bytes	reserved2	
5	r	8 bytes	TimeStamp	Number of microseconds from some fixed reference. The time interval between any two events can be found by the difference between their time stamps.
6	r	4 bytes	User Info	Additional user information. The structure of this data varies for different types of access (e.g., SCSI, ATA) and is well known in the industry.

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0180h
GroupType	SCALAR
Name	***OPTIONAL HISTORICAL STATS SUPPORT/CONTROL***
Description	Identifies optional historical statistics supported and controls the logging of the supported statistics.

FieldIdx	(r/w)	Field Size	Parameter Name	Description	
0	r/w	**1 byte**	StatisticsControl	Controls the logging of the optional statistics.	
				bit 0	Storage Statistics 0 - Disabled 1 - Enabled
				bit 1	Cache Statistics 0 - Disabled 1 - Enabled
				bits 2-7	Reserved
1	r	1 byte	reserved1		
2	r	2 bytes	reserved2		
3	r	4 bytes	StorageStatistics	Bit encoding (0-31) of the optional storage statistics supported. Each bit number corresponds to the same statistic attribute.	
				0	Statistic not supported
				1	Statistic supported
4	r	4 bytes	CacheStatistics	Bit encoding (0-31) of the optional cache statistics supported. Each bit number corresponds to the same statistic attribute.	
				0	Statistic not supported
				1	Statistic supported

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0181
GroupType	SCALAR
Name	***STORAGE HISTORICAL STATS*** - Optional
Description	Statistical counters to characterize the number of total read/write accesses and read/write accesses by size.

FieldIdx	**(r/w)**	**Field Size**	**Parameter Name**	**Description**
0	r	8 bytes	ReadCommands	Accumulated count of the number of read commands
1	r	8 bytes	WriteCommands	Accumulated count of the number of write commands
2	r	1 byte	DataUnit	Unit of measure for the I/O Range attributes, expressed as a power of two-bit count (0=1 bit; 3=1 byte; 4=16 bit word; 5=32 bit word)
3	r	1 byte	reserved1	
4	r	2 bytes	reserved2	
5	r	8 bytes	IORange1Read	Accumulated count of the data units read of the size of the data unit.
6	r	8 bytes	IORange2Read	Accumulated count of the data units read of the size range of two to three times the data unit.
7	r	8 bytes	IORange3Read	Accumulated count of the data units read of the size range of four to seven times the data unit.
8	r	8 bytes	IORange4Read	Accumulated count of the data units read of the size range of eight times the data unit, or greater.
9	r	8 bytes	IORange1Write	Accumulated count of the data units written of the size of the data unit.
10	r	8 bytes	IORange2Write	Accumulated count of the data units written of the size range of two to three times the data unit.
11	r	8 bytes	IORange3Write	Accumulated count of the data units written of the size range of four to seven times the data unit.
12	r	8 bytes	IORange4Write	Accumulated count of the data units written of the size range of eight times the data unit or greater.
13	r	8 bytes	NumberSeeks	Accumulated count of the seek commands issued.

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0182
GroupType	SCALAR
Name	***CACHE HISTORICAL STATS*** - Optional
Description	Historical statistics to characterize cache operational efficiency and failure logging.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	CacheAccess	Accumulated count of the times the cache has been accessed.
1	r	8 bytes	CacheHit	Accumulated count of the cache hits.
2	r	8 bytes	PartailCacheHit	Accumulated count of the partial cache hits.
3	r	8 bytes	HitDataSize	Indicates the number of bytes available in the cache when a partial hit occurred.
4	r	4 bytes	ValidUsage	Amount of cache currently holding valid data (in kilobytes).
5	r	4 bytes	DirtyUsage	Amount of cache data not yet written to medium (in kilobytes).
6	r	4 bytes	TimeLastFault	Number of seconds between power on and the last detected cache fault or failure.
7	r	4 bytes	LastFaultFailure	SCSI sense code associated with the most recent detected cache fault or failure.

Note: The following groups apply to a RAID topology that is mapped after the SCC (SCSI-3 Controller Commands) model.

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0200
GroupType	SCALAR
Name	***VOLUME SET INFORMATION*** - Optional
Description	Only applies to Block Storage SubClass = RAID Disk

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	64 bytes	Name	Name of the volume set.
1	r	8 bytes	TotalStorageCapacity	Total size of user data space (in bytes).
2	r	8 bytes	StripeLength	Number of ps_extents that form a user data stripe. This value is 0 except when the ps_extent group is used.
3	r	8 bytes	InterleaveDepth	Number of ps_extents to stripe as a collective set. This value is 0 except when the ps_extent group is used.

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0201
GroupType	SCALAR
Name	***PROTECTED SPACE EXTENT*** - Optional
Description	Only applies to Block Storage SubClass = RAID Disk

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	StartAddress	Starting LBA of the p_extent from which this ps_extent is derived.
1	r	8 bytes	NumberBlocks	Number of user data blocks in this ps_extent.
2	r	4 bytes	BlockSize	Size (in bytes) of the blocks that form this ps_extent.
3	r	4 bytes	DataStripeGranularity	The units in which the User Data Stripe Depth is given. 0 Other 1 Unknown 2 Bits 3 Bytes 4 16BitWords 5 32BitDoubleWords 6 Blocks
4	r	4 bytes	DataStripeDepth	The number of granularity units that form the stripe size for this ps_extent.

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0202
GroupType	SCALAR
Name	***AGGREGATE PROTECTED SPACE EXTENT*** - Optional
Description	Only applies to Block Storage SubClass = RAID Disk

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	NumberBlocks	The total user data blocks located on both a single storage device and part or all of a single volume set. The block size is determined by storage device association with this aggregate protect space extent. If no volume set is associated with an aggregate protect space extent, then this indicates the number of blocks available.

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0203
GroupType	SCALAR
Name	***PHYSICAL EXTENT*** - Optional
Description	Only applies to Block Storage SubClass = RAID Disk

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	StartAddress	The starting LBA on a storage device from which this p_extent is derived.
1	r	8 bytes	NumberBlocks	Total consecutive blocks contained in this p_extent.
2	r	4 bytes	BlockSize	Size (in bytes) of the blocks that form this p_extent. The value of zero shall represent a variable block size.
3	r	4 bytes	GranularityUnit	The units in which the following check data and user data fields are given. 0 Other 1 Unknown 2 Bits 3 Bytes 4 16BitWords 5 32BitDoubleWords 6 Blocks
4	r	8 bytes	CheckDataInterleave	Number of granularity units of user data to skip before starting the check data interleave.
5	r	8 bytes	CheckData	Number of consecutive granularity units to reserve for check data within the p_extent.
6	r	8 bytes	UserData	Number of consecutive granularity units to reserve for user data within the p_extent.

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0204
GroupType	SCALAR
Name	***AGGREGATE PHYSICAL EXTENT*** - Optional
Description	Only applies to Block Storage SubClass = RAID Disk

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	NumberBlocks	Total consecutive blocks (including check data) contained in this p_extent. The block size is determined by the storage device associated with this aggregate p_extent. Note: If no redundancy group is associated with an aggregate p_extent, then this indicates the number of p_extent blocks available.
1	r	8 bytes	CheckData	Number of blocks contained in this aggregate p_extent to be used as check data. If the aggregate p_extent is available, then this value should be zero.

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0205
GroupType	SCALAR
Name	***REDUNDANCY*** - Optional
Description	Only applies to Block Storage SubClass = RAID Disk

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	1 byte	RedundancyType	Specifies the type of the redundancy formed. 0 Other 1 Unknown 2 None 3 Copy 4 XOR 5 P+Q 6 S 7 P+S

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0206
GroupType	TABLE
Name	***COMPONENT SPARES*** - Optional
Description	Only applies to Block Storage SubClass = RAID Disk Non-data sparing.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	1 byte	RowNumber	Key for accessing this table.
1	r	1 byte	SpareType	Specifies the type of redundancy formed. 0 StorageController 1 BusPort
2	r	1 byte	ToBeSparedIndex	The key into the table identified by the spare type to reference the specific object to be spared.
3	r	1 byte	SparedIndex	The key into the table identified by the spare type to reference the specific object of the spare.
4	r	1 byte	SpareFunctioningState	Specifies the functioning state of the spare. 0x0 Other 0x1 Unknown 0x2 Inactive/Standby 0x3 Active/Standby 0x4 Load balancing/Active/Standby

Table 6-17. BSA Parameters Group (continued)

GroupNumber	0207
GroupType	TABLE
Name	***ASSOCIATION TABLE*** - Optional
Description	Only applies to Block Storage SubClass = RAID Disk The association table is used to represent the physical and logical topology.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	1 byte	RowNumber	Key for accessing this table.
1	r	1 byte	Type	Specifies the type of association being made. 00h Physical Organization 01h Logical Organization 02h Logical to Physical Organization 03h Protection Organization 04h Spare Organization 05h Cache Organization 06h Software Organization
2	r	1 byte	Object1Type	Type of object referred to by object 1 index. Combined with the object 1 index, forms a complete reference to a specific object. 00h Storage Controller 01h Storage Device 02h Bus Port 03h Volume Set 04h Protected Space Extent 05h Aggregate Protected Space Extent 06h Physical Extent 07h Aggregate Physical Extent 08h Redundancy Group 09h Cache 0Ah Software
3	r	1 byte	Object1Index	The key into the table specified by the object 1 type.
4	r	1 byte	Object2Type	Type of object referred to by object 2 index. Combined with the object 2 index, a complete reference is made to a specific object. 00h Storage Controller 01h Storage Device 02h Bus Port 03h Volume Set 04h Protected Space Extent 05h Aggregate Protected Space Extent 06h Physical Extent 07h Aggregate Physical Extent 08h Redundancy Group 09h Cache 0Ah Software
5	r	1 byte	Object2Index	The key into the table specified by the object 2 type.

6.5 Tape Storage Class

6.5.1 Operational Model

A tape storage drive provides sequential access to a permanent storage medium. The DDM registers a different tape class device for each logical drive it provides. The client, typically an OSM, performs tape storage operations by sending requests to, and listening for replies from, a tape class device.

In general, the term *device* applies to the abstract interface produced by the message class. The client sends messages to that device by indicating the TID for that device in the message's TargetAddress field. The terms *physical device* and *drive* refer to the physical device or facility the DDM controls. *Behavior* of the device refers to the software as well as its physical device(s).

The Tape Storage class operation is based on the Random Block Storage class (refer to section 6.4) in general structure. Although the structure is similar, there are several important differences in the operational model, and thus, the implementation. The following sections describe the operational model and key characteristics for tape devices. In addition, tape devices have different mechanical delay characteristics from their block storage counterpart, and thus have different timeout values. Section 6.5.1.7 defines timeout values for the tape device commands.

6.5.1.1 Sequential Media Access

Tape devices are based on sequential media access where command order is rigidly maintained for all read, write and access operations. Thus, order must be guaranteed for all operations.

6.5.1.2 Variable/Fixed Block Support

Tape drives support both fixed and variable block addressing. In variable block mode, requests specify a separate byte count for each operation. In this mode, the block size changes from one block to the next. In fixed block mode, the request specifies the number of blocks to transfer, assuming that the fixed block size is already initialized. In fixed block mode, the block size of a particular tape usually remains the same, although devices may allow changing fixed block sizes in the middle of the tape. Also, devices may support intermixing variable and fixed block modes on a tape. A wide range of restrictions and flexibility apply to different devices in this area.

6.5.1.3 Compression and Write Density

Current generation tape drives often support data compression specific to a drive type or its manufacturer. Also, many tape devices support a software/interface control for writing older, lower-capacity formats to a tape. In these cases, the tape usually contains device-accessible information at Beginning of Tape (BOT) that specifies the compression status and write density. In most cases, write density can only be selected at BOT. For any given tape device, compression may change in mid tape, or be selected only at BOT.

For read operations, tape devices universally detect the compression status and write density of the installed tape medium, and perform appropriate read operations for the installed tape.

6.5.1.4 File Marks, Set Marks, and Partitions

File marks and set marks are used for two functions. First, they provide a hierarchical structure that can speed tape access. In this case, file marks are the lower portion of the hierarchy, and set marks are hierarchically superior. Tape data is generally structured by an application or OS with file and set marks at critical locations for either for performance or function. Tape applications are then written using the designed file mark/set mark structure. Filemarks are universally supported on tape drives, and set marks tend to be more device dependent.

The second use for file marks and set marks is data transfer synchronization. In tapes, the I/O interface is usually substantially faster than the mechanical tape interface, particularly regarding start/stop delays. To improve performance, most tape write operations let the device return a completion status when the write data is in the device cache. Later, the device controls when it actually writes the cached data to tape. Writing a file mark or set mark is the typical way to force a flush of cache to tape. Also, file and set marks provide a natural checkpoint in case of subsequent media failures.

Partitions are above file and set marks in the hierarchy. Partitions are typically device dependent, and tie closely to the device and media physical characteristics.

6.5.1.5 Error Types and Reporting

Several specific error situations occur in the tape model. Different than disks, tape applications and I/O subsystems must anticipate and routinely handle these errors.

6.5.1.5.1 Positioning/ILI Errors

Some applications allow foreign formats for which the tape’s block size is unknown. These applications have established algorithms that allow them to read unknown tapes. The ILI, positioning errors, and related status information are critical to this type of application. As a result, ILI and positioning errors are not necessarily medium or device failures, but are important tape application tools.

6.5.1.5.2 Media Errors

Tape media failures stem from several sources. Those include worn media, dirty heads, extreme environmental conditions and a faulty device. The experienced tape user expects some media errors, and the appropriate response is context dependent. The operator may clean the heads and retry any failed operation. If it is a write tape operation (backup), the operator tends to discard a failed medium to eliminate a marginal component from the backup set. For read tape operations (restore), the operator often makes extensive efforts to read a tape if it fails, typically trying the restore on other compatible devices, if available. Modern tape drives give users and management software more information and tools to allow intelligent management of media. This class definition supports all information in this area, and anticipates further improvements in predictive capabilities from future tape devices.

6.5.1.5.3 Queue Status on Error

When an error occurs and multiple requests are outstanding or cached writes are being used, the application should refer to the last managed checkpoint where media flush was successfully forced. The safest application behavior is clearing the queue, and rewinding/removing the current medium. Tape devices vary in their support of subsequent access operations after a command fails. Unless specific devices are well characterized and understood, operations other than rewind are problematic.

6.5.1.6 Tape Devices Only

Libraries and Changers not Covered: This class deals strictly with controlling tape devices, not tape libraries or changers.

6.5.1.7 Timing Out Requests

Many tape storage requests have some form of associated timeout. Timeout values are accessible via a ***UtilParamsGet*** (GROUP=**Operational_Control**) request (see section 6.5.5). Request functions have some form of associated multiplier or formula to determine the timeout value, as stated in Table 6-18. Some functions provide a TimeMultiplier in the request that multiplies the timeout for that request. A TimeMultiplier of zero suspends the timeout for that transaction.

Timeout policy is enforced at the DDM level when the DDM initiates an action. The client may have a separate timeout policy.

Table 6-18. Timeout Formula

Message	Timeout Formula
TapeCacheFlush	TimeMultiplier x (ShortPositionTimeout + RWTimeout)
TapeCmprsnSet	TimeMultiplier x ShortPositionTimeout
TapeDataErase	TimeMultiplier x EraseTimeout
TapeDataRead	Fixed Mode - TimeMultiplier x (ShortPositionTimeout + (RecordCount x RWTimeout))
	Variable Mode - TimeMultiplier x (ShortPositionTimeout + RWTimeout)
TapeDataWrite	Fixed Mode - TimeMultiplier x (ShortPositionTimeout + (RecordCount x RWTimeout))
	Variable Mode - TimeMultiplier x (ShortPositionTimeout + RWTimeout)
TapeDataWriteVerify	Fixed Mode - 2 x TimeMultiplier x (ShortPositionTimeout + (RecordCount x RWTimeout))
	Variable Mode - TimeMultiplier x (ShortPositionTimeout + RWTimeout)
TapeDensitySet	TimeMultiplier x ShortPositionTimeout
TapeDeviceReset	TimeMultiplier x (ShortPositionTimeout + RWTimeout)
TapeMarksWrite	TimeMultiplier x (ShortPositionTimeout + RWTimeout)
TapeMediaEject	TimeMultiplier x MediaMovementTimeout
TapeMediaLoad	TimeMultiplier x MediaMovementTimeout
TapeMediaLock	TimeMultiplier x ShortPositionTimeout
TapeMediaPosition	TimeMultiplier x LongPositionTimeout
TapeMediaUnlock	TimeMultiplier x ShortPositionTimeout
TapePartitionCreate	TimeMultiplier x LongPositionTimeout
TapePowerMgt	TimeMultiplier x PowerDownTimeout
TapeStatusCheck	TimeoutBase

If TimeMultiplier is zero, then do not timeout.

6.5.2 Tape Storage Reply Messages

The reply message structure and behavior is generally the same as described for Random Block Storage in section 6.4.4. The reply messages for tape storage are based on those for Random Block Storage, except that the DetailedStatusCode is divided into two fields: a TapeStatusCode, as specified in Table 6-20, and a four-bit tape positioning status (TapePos), as specified in Table 6-19.

Note: The DDM never sets the FAIL bit in the MessageFlags field. If the DDM receives a request with an unknown Function code or an ill-formed message, it replies with a ***Transaction Error Reply Message*** as specified in Chapter 3.

When the DDM aborts a request because of system state changes, it sends a final reply for each outstanding transaction as an error.

6.5.2.1 Tape Storage Status Codes

Table 6-19. Tape Position (TapePos) Field definition

Bit	Name	Definition
12	ILI	If the ILI bit is set, the current request results in a data transfer, but the requested transfer length does not equal the transfer length recorded on the tape. The TransferCount field gives the number of bytes actually transferred, which is the lesser of the requested transfer and the block length on the tape medium. The DDM only sets the ILI bit in reply messages containing an error status.
13	BOT	If the BOT bit is set, the current request completed at BOT (beginning of tape).
14	EOT	If the EOT bit is set, the current request completed at EOT (end of tape).
15	MK	If the MK bit is set, the current request completed in a filemark or set mark position.

Table 6-20. Tape Detailed Status Code

TapeStatusCode (*TAPE_STATUS*_xxx*)*	Description
_ACCESS_VIOLATION	The device is locked for exclusive access by another party.
_BOT	Error status when tape position is at beginning of tape, preventing successful completion of the Request.
_BUS_ERROR	Problem detected with the operation of the device's bus, but operation completed with successful retry attempts. RetryCount supplied in Successful Completion Reply Message. Reading the Error Log provides hardware-specific status.
_BUS_FAILURE	The operation failed due to a problem with the device's bus.
_DEFERRED_ERROR	Error occurred during a write to media for an unspecified past write command, where successful status returned when control flags were set to write-to-cache mode (see 0).
_DEVICE_FAILURE	Device does not respond or responds with fault.

_DEVICE_NOT_READY The device is not ready for access. The client may determine the device's state by using the ***UtilParamsGet*** request with GROUP = DEVICE_INFO and/or GROUP = POWER_CTL.

_DEVICE_RESET After a reset, all requests aside from utility messages (base class requests) are returned with *DEVICERESET* status until the reset is acknowledged by the user.

_EOD Error status when tape position is at the end-of-data area, preventing successful completion of the Request.

_EOT Error status when tape position is the end of tape, preventing successful completion of the Request.

_FILEMARK_DETECTED Error status when device encountered a file mark during an access operation, preventing successful completion of the Request. Tape position is past the detected file mark in the direction of tape motion for the associated request.

_ILLEGAL_BLOCK_TRANSFER Attempt to perform a fixed-block request when BlockSize parameter of DEVICE_ID group is set to zero.

_ILLEGAL_COMPRESSION_CONTROL Error report condition caused when a ***TapeCmprsnSet*** request occurs, when disallowed by device state or tape position.

_ILLEGAL_LENGTH_INDICATION Error Report condition caused by read operation with different record size than the media record size. Examine TransferCount of Error Log (Table 6-35) to determine the number of bytes transferred.

_ILLEGAL_LOAD_OPERATION Error report condition caused by an attempt to request a load, unload, lock or unlock the device does not support, or is appropriate to the current device state.

_ILLEGAL_SET_DENSITY_REQUEST Error report condition caused when a ***TapeDensitySet*** request occurs that is illegal on a specific device or when disallowed by the current device state or tape position.

_LOAD/UNLOAD_FAILURE Error occurred during a load or unload

_MEDIA_ERROR Media retry. Device was forced to retry to read/write the data. Retry count supplied. GET_LAST_LOGGING_DATA can retrieve hardware-specific status.

_MEDIA_FAILURE The operation failed due to an error on the medium.

_MEDIA_LOCKED Medium locked by another party.

_MEDIA_NOT_PRESENT Removable medium not loaded.

_POWER_RESET_DETECTED A power cycle or device/bus reset was detected.

_PROTOCOL_FAILURE The operation failed due to a communication problem with the device.

_SETMARK_DETECTED Error status when device encountered a set mark during an access operation, preventing completion of the request. Tape position is past the detected set mark in the direction of tape motion for the associated request.

_SUCCESS_UNCONDITIONAL Successful operation - no reportable retries or exceptions.

_SUCCESS_WITH_RETRIES Successful operation - retries required. RetryCount supplied available in Successful Completion Reply Message.

	GET_LAST_LOGGING_DATA can retrieve hardware-specific status.
_TIMEOUT	The operation failed because the time-out value specified for this request has been exceeded.
_UNSUPPORTED_OPERATION	The device received a request for an unsupported operation.
_VOLUMECHANGED	After a volume change, all requests aside from utility messages (base class requests) return with *VOLUMECHANGED* status until the event is acknowledged.
_WRITE_PROTECTED	The medium is write protected or read only.

Note that the above TapeStatusCode values do not apply equally to successful transfers, errors, and aborts. The DDM must report an appropriate code.

6.5.2.2 Successful Completion

For requests that completed without errors, the ReqStatus is set to reflect successful completion, and the ReplyPayload indicates the total bytes transferred.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		1 1 0 0 0 0 x 0	VersionOffset = 01h	0
Function	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
_SUCCESS	RetryCount	TapePos	TapeStatusCode	16 (24)
TransferCount				20 (28)

offset values for 64-bit context field size in ()

Figure 6.38. Successful Completion Reply Message for Tape Storage Class

Fields

RetryCount	Number of retries to complete the request. A value of zero means success on the first try. Reason for unsuccessful attempts provided in the TapeStatusCode.
TapePos	Bit-specific field as described in Table 6-19.
TapeStatusCode	If non-zero, contains a warning code describing the nature of the recovered operation, as described in Table 6-19.
TransferCount	The number of bytes transferred.

6.5.2.3 Aborted Operation

Transactions aborted at the request of the originator have the ReqStatus set to *STATUS_CODE_ABORT_NO_DATA_TRANSFER*, *STATUS_CODE_ABORT_PARTIAL_TRANSFER* or *STATUS_CODE_ABORT_DIRTY* and there is no ReplyPayload.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		1 1 0 0 0 0 x 0	VersionOffset = 01h	0
Function	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
AbortCode	RetryCount	TapePos	TapeStatusCode	16 (24)

offset values for 64-bit context field size in ()

Figure 6.39. Aborted Operation Reply Message for Tape Storage Class

AbortCode Contains a generic status code, as specified in chapter 3 (*STATUS_CODE_ABORT_NO_DATA_TRANSFER*, *STATUS_CODE_ABORT_PARTIAL_TRANSFER* or *STATUS_CODE_ABORT_DIRTY*).

6.5.2.4 Progress Reports

Progress replies are for requests that can take a relatively long time to complete, such as verifying operation of an entire device. Progress replies are appropriate only for certain requests, as indicated by the ProgressReport bit in the request's ControlFlags field. When enabled, the DDM sends periodic progress reports. The ReqStatus value of *STATUS_CODE_PROGRESS_REPORT* identifies a progress report. Progress replies are always followed by a reply with a final ReqStatus (e.g., *STATUS_CODE_SUCCESS, STATUS_CODE_ABORT_NO_DATA_TRANSFER*, *STATUS_CODE_ABORT_PARTIAL_TRANSFER*, *STATUS_CODE_ERROR_NO_DATA_TRANSFER*, *STATUS_CODE_ERROR_PARTIAL_TRANSFER, STATUS_CODE_ABORT_DIRTY, STATUS_CODE_ERROR_DIRTY*). The exception is when the progress report is returned in response to a ***TapeStatusCheck*** request. In this case, the reply is a final message.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		1 0 0 0 0 0 x 0	VersionOffset = 01h	0
Function	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
_Progress_Report	RetryCount	0 0 0 0	000h	16 (24)
Reserved			PercentComplete	20 (28)

offset values for 64-bit context field size in ()

Figure 6.40. Progress Report Reply Message for Tape Storage Class

Fields

PercentComplete The percentage complete value, 0-100

RetryCount Indicates the number of retries attempted on the current operation.

Progress is measured in 1% resolution. The DDM determines the rate at which these messages are sent, although a rate of one progress message per second is a good guideline.

A ***StatusCheck*** message is an alternative to enabling progress reporting (see section 6.5.4.17).

6.5.2.5 Error Reports

For requests that do not complete successfully, the ReqStatus indicates *STATUS_CODE_ERROR_NO_DATA_TRANSFER*, *STATUS_CODE_ERROR_PARTIAL_TRANSFER*, or *STATUS_CODE_ERROR_DIRTY* and the TapeStatusCode contains the detailed error code.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		1 1 0 0 0 0 x 0	VersionOffset = 01h	0
Function	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
ReqStatus	RetryCount	TapePos	TapeStatusCode	16 (24)
TransferCount				20 (28)

offset values for 64-bit context field size in ()

Figure 6.41. Unsuccessful Completion Reply Message for Tape Storage Class

Fields

ReqStatus — Contains a generic status codes as specified in chapter 3 (*STATUS_CODE_ERROR_NO_DATA_TRANSFER*, *STATUS_CODE_ERROR_PARTIAL_TRANSFER*, or *STATUS_CODE_ERROR_DIRTY*).

TapeStatusCode — Contains as specific a code as possible that describes the nature of the failure (Table 6-20).

6.5.3 Support for Utility Messages

6.5.3.1 Lock and Lock Release

The ***UtilLock*** request causes the DDM to guarantee the initiator exclusive access to the device until a ***UtilLockRelease*** request is issued by the same initiator. See the ***UtilLock*** message in 6.1.3.14. Also see ***UtilDeviceReserve*** message for acquiring rights to a device with multiple paths.

A ***UtilLockRelease*** request cancels a previous reservation. Issuing a ***UtilLockRelease*** request without completing a ***UtilLock*** request causes the DDM to fail the request.

6.5.3.2 Event Notification for Tape Storage Devices

Each Tape Storage class device supports the generic events specified in section 6.1.3.4 and the ***UtilEventRegister*** request. In addition, each device supports the following tape storage events:

Table 6-21. Tape EventIndicator Assignments

Event Name	Bit	Description
VolumeLoad	0	New medium has been loaded into the device
VolumeUnload	1	The medium on the device has been unloaded
VolumeUnloadRequest	2	An external unload request and medium is locked. The client must unlock the medium before the unload request can be honored.
SCSI_SMART	4	Reports SCSI SMART data is received

Table 6-22. EventData for Tape Events

Event Name	EventData
VolumeLoad	No data
VolumeUnload	No data
VolumeUnloadRequest	No data
SCSI_SMART	SCSI ASC and ASCQ (2 bytes)

6.5.3.3 Getting and Setting Parameters

Both the client (service user) and management use ***UtilParamsGet*** and ***UtilParamsSet*** utility messages (section 6.1.3) to read and modify parameter for tape devices. The list of parameter groups and their format for Tape Storage class devices is specified in section 6.5.5.

6.5.4 Tape Storage Request Messages

Table 6-23 lists requests that a client can make to a Tape Storage class device. The following sections define these requests in alphabetic order.

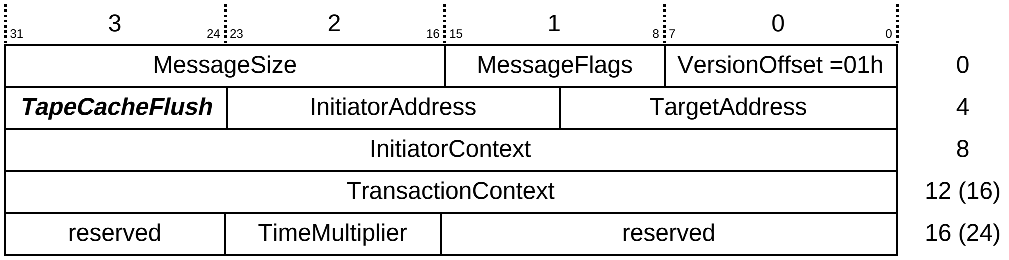
Table 6-23. Request Messages for the Tape Storage Class

Function code	Description
TapeCacheFlush	Write cached data to medium
TapeCmprsnSet	Control compression during write operations
TapeDataErase	Erase tape medium
TapeDataRead	Read number of bytes or blocks from current location
TapeDataWrite	Write number of bytes or blocks at current location
TapeDataWriteVerify	Write number of bytes or blocks at current location and verify transfer to media
TapeDensitySet	Set the write density for the current medium
TapeDeviceReset	Reset the device
TapeMarksWrite	Write file marks or set marks
TapeMediaEject	Eject tape medium from device
TapeMediaLoad	Load tape medium into device
TapeMediaLock	Lock tape medium into device
TapeMediaPosition	Position read/write heads
TapeMediaUnlock	Unlock tape medium in device
TapePartitionCreate	Create a tape partition
TapePowerMgt	Power management
TapeStatusCheck	Check device status

All Tape Storage class messages are single-transaction messages. Typically, the MessageFlags field for requests should be set to 00h (for 32-bit context size) or 02h (for 64-bit context size). For a normal completion reply, the MessageFlags field should contain C0h (for 32-bit context size) or C2h (for 64-bit context size). Since some requests provide an SGL the value of the VersionOffset field depends on the location of the SGL. Since all replies are single-transaction replies, the VersionOffset field should be set to 01h for all replies.

6.5.4.1 Cache Flush Message

The ***TapeCacheFlush*** request causes the DDM to write all dirty cached data to the medium. The DDM must not issue a completion status reply until all data is actually on the medium. The DDM must ensure that any data cached downstream also flushes to the medium.



31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset =01h	0
TapeCacheFlush	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	reserved		16 (24)

offset values for 64-bit context field size in ()

Figure 6.42. *TapeCacheFlush* Request Message

A failure should only be reported if the medium is no longer accessible or an uncorrectable error occurs.

6.5.4.2 Compression Set Message

The ***TapeCmprsnSet*** request controls whether data compression is used in subsequent write operations. If the CompControl is 0, the device disables data compression in subsequent write operations to the current partition. If the CompControl is 1, the device enables data compression in subsequent write operations to the current partition. If the device cannot change compression modes in its current state or tape position, it returns an *ILLEGAL_COMPRESSION_CONTROL* in TapeStatusCode (Table 6-20). For devices supporting multiple tape partitions, the effect of this command on other partitions is undefined and device dependent.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset =01h	0
TapeCmprsnSet	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
reserved		ControlFlags		16 (24)

offset values for 64-bit context field size in ()

Figure 6.43. *TapeCmprsnSet* Request Message

Fields:

ControlFlags — Options for performing this command.

Bit 0: CompControl - When this bit is set, compression is on. When this bit is cleared, compression is off.

Other bits reserved

6.5.4.3 Data Erase Message

The ***TapeDataErase*** request directs the DDM to erase data on the current medium. The device erases all data on the medium from the current position to the end of the tape. If the SecureFlag is 0, the DDM performs the minimum necessary to clear the medium (e.g., reformat). If the SecureFlag is 1, the DDM performs a complete erase of all data blocks on the tape.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset =01h	0
TapeDataErase	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	ControlFlags		16 (24)

offset values for 64-bit context field size in ()

Figure 6.44. *TapeDataErase* Request Message

Fields

ControlFlags — Specifies specific options for performing this command.

Bit 0: SecureFlag - When this bit is set, the DDM must assure that the old data is wiped from the medium.

Bit 7: ProgressReport - When this bit is set, the device posts Progress Reports (see section 6.5.2.4). When it is not set, the device does not post Progress Reports.

Other bits reserved

6.5.4.4 Data Read Message

The ***TapeDataRead*** request causes the DDM to read TransferCount number of bytes or blocks from the current tape location to a buffer described by the SGL. ControlFlags value specifies whether the TransferCount is the number of bytes or blocks. If block transfers are specified, the operation returns an error if the value of the BlockSize parameter (DEVICE_INFORMATION group, Table 6-27) is zero.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
TapeDataRead	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	ControlFlags		16 (24)
TransferCount				20 (28)
SGL				24 (32)
				n

offset values for 64-bit context field size in ()

Figure 6.45. *TapeDataRead* Request Message

Fields

ControlFlags — Options for performing this command.

Bit 5: DataMode - When this bit is cleared, the TransferCount defines number of blocks to transfer (Block mode). When set, the operation is variable mode, and TransferCount defines number of bytes to transfer.

Other bits reserved.

TransferCount — Depending on DataMode, this field specifies the number of blocks or bytes to transfer.

VersionOffset — A value of 61h for 32-bit context size and 81h for 64-bit context size.

6.5.4.5 Data Write Message

The ***TapeDataWrite*** request causes the DDM to write TransferCount number of bytes or blocks to the current tape location from a buffer described by the SGL. ControlFlags value specifies whether the TransferCount is the number of bytes or blocks. If block transfers are specified, the operation returns an error if the value of the BlockSize parameter (DEVICE_ INFORMATION group, Table 6-27) is zero.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
TapeDataWrite	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	ControlFlags		16 (24)
TransferCount				20 (28)
SGL				24 (32)
				n

offset values for 64-bit context field size in ()

Figure 6.46. *TapeDataWrite* Request Message

Fields

ControlFlags — Specifies specific options for performing this command.

Bit 0: NoRetry - When this bit is set, the request does not retry if it fails.

Bit 3: WriteThruCache - When this bit is set, the DDM must make the data durable before the request completes.

Bit 4: WriteToCache - When this bit is set, the DDM can reply before the request is durable on the medium, assuming the data is cached.

Bit 5: DataMode - When this bit is cleared, the TransferCount defines number of blocks to transfer (Block mode). When set, the operation is variable mode, and TransferCount defines number of bytes to transfer.

Other bits reserved.

TransferCount — Depending on DataMode, this field specifies the number of blocks or bytes to transfer.

VersionOffset — A value of 61h for 32-bit context size and 81h for 64-bit context size.

6.5.4.6 Data Write Verify Message

The ***TapeDataWriteVerify*** request functions just like the ***TapeDataWrite*** request, except that the DDM must verify the transfer's accuracy. The DDM must write through any cache to conclude this operation.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
TapeDataWriteVerify	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	ControlFlags		16 (24)
TransferCount				20 (28)
SGL				24 (32) *n*

offset values for 64-bit context field size in ()

Figure 6.47. *TapeDataWriteVerify* Request Message

Fields

ControlFlags — Options for performing this command.

Bit 0: NoRetry - When this bit is set, the request should not be retried if it fails.

Bit 5: DataMode - When this bit is cleared, the TransferCount defines number of blocks to transfer (Block mode). When set, the operation is variable mode, and TransferCount defines number of bytes to transfer.

Other bits reserved

TransferCount — Depending on DataMode, this field specifies the number of blocks or bytes to transfer.

VersionOffset — A value of 61h for 32-bit context size and 81h for 64-bit context size.

6.5.4.7 Density Set Message

The ***TapeDensitySet*** request changes the write density of the current medium. Support for this request is drive dependent. In general, the DDM passes the density code to the drive and reports status, based on its ability to pass that information.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
TapeDensitySet	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
Density	TimeMultiplier	Reserved		16 (24)

offset values for 64-bit context field size in ()

Figure 6.48. *TapeDensitySet* Request Message

Fields

Density The density code is a drive-specific parameter that controls the write density of the current medium. The contents of this field and its ability to change density are drive specific.

6.5.4.8 Device Reset Message

The ***TapeDeviceReset*** request returns the device to a known state. The DDM aborts all outstanding requests (i.e., those queued for the device, as well as those in progress) and returns them with an appropriate abort status.

3 31 … 24	2 23 … 16	1 15 … 8	0 7 … 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
TapeDeviceReset	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	ControlFlags		16 (24)

offset values for 64-bit context field size in ()

Figure 6.49. *TapeDeviceReset* Request Message

Fields:

ControlFlags Options for performing this command.

Bit 0: HardReset - When this bit is set, the DDM performs a hard reset and when it is cleared, the DDM performs a soft reset. In general, a hard reset initializes the device, and a soft reset clears the protocol to the medium.

Bit 7: ProgressReport - When this bit is set, the device posts Progress Reports (see section 6.5.2.4). When it is not set, the device does not post Progress Reports.

Other bits reserved

6.5.4.9 Marks Write Message

The ***TapeMarksWrite*** request causes the DDM to write TransferCount number of file or set marks to the tape at the current location. If TransferCount equals zero, the result equals a Flush request and the tape position stays the same.

31 **3** 24	23 **2** 16	15 **1** 8	7 **0** 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
TapeMarksWrite	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	ControlFlags		16 (24)
TransferCount				20 (28)

offset values for 64-bit context field size in ()

Figure 6.50. *TapeMarksWrite* Request Message

Fields

ControlFlags
Options for performing this command.

Bit 0: MarkType - When this bit is cleared, file marks are written. When set, the device writes set marks.

Bit 5: DataMode - When this bit is cleared, the TransferCount defines the number of blocks to transfer (Block mode). When set, the operation is in variable mode, and TransferCount defines the number of bytes to transfer.

Bit 7: ProgressReport - When this bit is set, the device posts Progress Reports (see 6.5.2.4). When it is not set, the device does not post Progress Reports.

Other bits reserved.

TransferCount
Depending on MarkType, this field specifies the number of file marks or set marks to write.

6.5.4.10 Media Eject Message

The ***TapeMediaEject*** request causes the device to eject the medium from the drive. If the device is locked from a ***TapeMediaLock*** request, the device fails any ***TapeMediaEject*** requests until the device receives a ***TapeMediaUnlock*** request. A failure also results if this device does not support the eject operation, as specified in Table 6-27.

31 **3** 24	23 **2** 16	15 **1** 8	7 **0** 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
TapeMediaEject	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
MediaIdentifier*				16 (24)

offset values for 64-bit context field size in ()

Figure 6.51 *TapeMediaEject* Request Message

* Media identifier is an optional parameter. It is currently set at -1, but may change when multiple tape devices are supported.

6.5.4.11 Media Load Message

When the DDM receives the ***TapeMediaLoad*** request, it attempts to load the medium identified by the MediaIdentifier. If the device cannot automatically load media, it returns an *UNSUPPORTED OPERATION* status. If the device supports lock, and the LockMedia bit is set, the DDM locks the medium in the device.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
TapeMediaLoad	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
Reserved	Reserved	ControlFlags		16 (24)
MediaIdentifier				20 (28)

offset values for 64-bit context field size in ()

Figure 6.52. *TapeMediaLoad* Request Message

Fields

ControlFlags — Options for performing this command.

Bit 0: LockMedia - When this bit is set, the device locks the medium in the device. When cleared, the device does not lock the medium.

Other bits reserved

Media Identifier — Media identifier is an optional parameter. It is currently set at -1, but may change when multiple tape devices are supported.

6.5.4.12 Media Lock Message

The ***TapeMediaLock*** request causes the DDM to lock the tape medium into the device. Once locked, a tape cannot be removed from the device, manually or under software control, until the DDM receives a ***TapeMediaUnlock*** request. If the device cannot lock the tape, it returns an *UNSUPPORTED OPERATION* status (Table 6-20).

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
TapeMediaLock	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
MediaIdentifier*				16 (24)

offset values for 64-bit context field size in ()

Figure 6.53. *TapeMediaLock* Request Message

Media identifier is an optional parameter. It is currently set at -1, but may change when multiple tape devices are supported.

6.5.4.13 Media Position Message

The ***TapeMediaPosition*** request causes DDM to position the medium to its beginning, end, or somewhere in between, depending on the PositioningMode byte.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
TapeMediaPosition	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
PositioningMode	TimeMultiplier	ControlFlags		16 (24)
Distance				20 (28)
(64-bits)				24 (32)

offset values for 64-bit context field size in ()

Figure 6.54. *TapeMediaPosition* Request Message

Fields

ControlFlags — Options for performing this command.

Bit 0: Setmarks - When this bit is cleared, a Mark operation is on file marks. When set, the operation is on set marks.

Bit 7: ProgressReport - When this bit is set, the device posts Progress Reports (see section 6.5.2.4). When it is not set, the device does not post Progress Reports.

Other bits reserved

PositioningMode — Specifies the positioning operation, as in Table 6-24.

Table 6-24. Media Positioning Modes

Value	Positioning Mode	Description
00h	BYTES_ABSOLUTE	Position read/write heads at Logical Byte Address (absolute) = Distance. A value of 0 positions the read/write heads at the beginning of the tape. A value of -1 positions the read/write heads at the end of the tape.
01h	BYTES_RELATIVE	Position read/write heads at Distance number of bytes relative to Current Location. The Distance is a signed value. Specifying a positive value moves the tape forward, and a negative value moves the tape backward.
02h	FILES_ABSOLUTE	Position read/write heads at Distance number of file partitions from the start of tape (absolute). A value of 0 positions the read/write heads at the beginning of the tape. A value of -1 positions the read/write heads at the end of the tape.
03h	FILES_RELATIVE	Position read/write heads at + Distance (relative) number of file partitions from the Current Partition. Distance is a signed value. Specifying a positive value moves the tape forward, and a negative value moves the tape backward. A value of zero positions the tape at the beginning of the current partition.
04h	BLOCKS_ABSOLUTE	Position read/write heads at Distance number of blocks (records) from the start of tape (absolute). A value of 0 positions the read/write heads at the beginning of the tape. A value of -1 positions the read/write heads at the end of the tape.
05h	BLOCKS_RELATIVE	Position read/write heads at + Distance (relative) number of blocks (records) from the current record. Distance is a signed value. Specifying a positive value moves the tape forward, and a negative value moves the tape backward. A value of zero positions the tape at the beginning of the current record.
06h	MARKS_ABSOLUTE	Position read/write heads at Distance number of file marks or set marks from the start of tape (absolute). A value of 0 positions the read/write heads at the beginning of the tape. A value of -1 positions the read/write heads at the end of the tape.
07h	MARKS_RELATIVE	Position read/write heads at + Distance (relative) number of filemarks or set marks from the current file. Distance is a signed value. Specifying a positive value moves the tape forward, and a negative value moves the tape backward. A value of zero positions the tape at the beginning of the current record.
08h	MARKS_CONSECUTIVE	Position the heads after the last of consecutive filemarks or set marks. The distance field specifies the amount of consecutive filemarks required for successful completion. Distance is a signed value. Specifying a positive value moves the tape forward, and a negative value moves the tape backward. A value of zero positions the tape at the beginning of the current record.
09h	PARTITIONS_ABSOLUTE	Position read/write heads at distance number of partitions from the start of tape (absolute). A value of 0 positions the read/write heads at the beginning of the tape.
FEh	END_OF_MEDIA	Position tape at End of Media

Value	Positioning Mode	Description
FFh	END_OF_DATA	Position tape at End of Data

6.5.4.14 Media Unlock Message

The ***TapeMediaUnlock*** request causes the DDM to unlock the drive so the tape can be removed from the device, manually or under software control. If the device does not support locking, the DDM returns an *UNSUPPORTED OPERATION* status (Table 6-20).

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
TapeMediaUnlock	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
MediaIdentifier*				16 (24)

offset values for 64-bit context field size in ()

Figure 6.55. *TapeMediaUnlock* Request Message

Media identifier is an optional parameter. It is currently set at -1, but may change when multiple tape devices are supported.

6.5.4.15 Partition Create Message

The ***TapePartitionCreate*** request causes the device to create the requested number of partitions on the tape. Each partition has an associated length field that describes the size of the partition in bytes. Devices may round approximate requested addresses and capacities. If the device supports only fixed parameters, as specified in the DeviceCapabilitySupport field of Table 6-27, the ***TapePartitionCreate*** request causes fixed partitioning and the DDM ignores the PartitionData.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
TapePartitionCreate	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
reserved	TimeMultiplier	ControlFlags		16 (24)
SGL :				20 (28)

offset values for 64-bit context field size in ()

Figure 6.56. *TapePartitionCreate* Request Message

Fields

ControlFlags Options for performing this command.

Bit 7: ProgressReport - When this bit is set, the device posts Progress Reports (see section 6.5.2.4). When it is not set, the device does not post Progress Reports.

Other bits reserved

SGL — Specifies a buffer containing the PartitionData. The SGL Immediate Data element (see Chapter 3) allows including the PartitionData in the message frame.

VersionOffset — A value of 51h for 32-bit context size and 71h for 64-bit context size.

31 3 24	23 2 16	15 1 8	7 0 0	
FunctionFlags	PartitionCount (n)			0
reserved				4
PartitionSize 1				8
(64-bit)				12
PartitionSize 2				16
(64-bit)				20
PartitionSize n				
(64-bit)				

Figure 6.57. Partition Data

Fields

FunctionFlags — Options for performing this command. None is yet defined.

PartitionCount — The number of PartitionSize elements.

PartitionSize — Each specifies the number of bytes in the partition. Partitions are created in the order listed.

6.5.4.16 Power Management Message

The ***TapePowerMgt*** request modifies the operational state of the device.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
TapePowerMgt	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
PwrAction	TimeMultiplier	ControlFlags		16 (24)

offset values for 64-bit context field size in ()

Figure 6.58. *TapePowerMgt* Request Message

Fields

ControlFlags — Options for performing this command.

Bit 7: ProgressReport - When this bit is set, the device posts Progress Reports (see section 6.5.2.4). When it is not set, the device does not post Progress Reports.

Other bits reserved

PwrAction — The DDM performs the specified power management operations described in Table 6-25.

Table 6-25. Tape Power Management Actions

Value	Description
01h	Power up partial – power up the device in a minimal state. The DDM need not load or tension tape media and/or heads.
02h	Power up – power the device up completely.
03h	Power up, load – power up the device completely and load media, if present.
20h	Quiesce device – flush any volatile state out to the volume and quiesce device activity.
21h	Power down partial – power down the device to a minimal state.
22h	Power down partial, unload – as above, but unload the volume, if removable.
23h	Power down, unload – fully power down the device, unloading the volume, if present.
24h	Power down, retain – fully power down the device, retaining the media.
other values	reserved

6.5.4.17 Status Check Message

Issuing a ***TapeStatusCheck*** request to a device returns either *STATUS_CODE_SUCCESS, STATUS_CODE_PROGRESS_REPORT*, or *STATUS_CODE_ERROR_NO_DATA_TRANSFER*. A *STATUS_CODE_SUCCESS* status indicates that the device is on-line and operating, while *STATUS_CODE_ERROR_NO_DATA_TRANSFER* indicates that the device may not be operating, depending on the detailed error code in the reply. (See section 6.4.4) A single *STATUS_CODE_PROGRESS_REPORT* reply is returned when the device is currently involved in a long operation, such as erase or rewind.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
TapeStatusCheck	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)

offset values for 64-bit context field size in ()

Figure 6.59. *TapeStatusCheck* Request Message

The reply for a ***StatusCheck*** depends on the state of the device. The table below describes replies:

Table 6-26. Status Check Replies

ReqStatus (*STATUS_CODE*_xxx*)*	**Meaning**
_ABORT_NO_DATA_TRANSFER	The ***StatusCheck*** request was aborted. The format of this message is an Abort Report (6.5.2.3)
_ERROR_NO_DATA_TRANSFER	The medium is not available. Determine the reason by interpreting the detailed status information. The format of this message is an Error Report (6.5.2.5)
_PROGRESS_REPORT	A request is currently active against the device that supports PROGRESS replies (***TapeMediaPosition***, ***TapeDataErase***, ***TapeMediaPosition***, etc.) The format of this message is a Progress Report (6.5.2.4)
_SUCCESS	The medium is available.

6.5.5 Managing Parameters of Tape Devices

Both the client (service user) and management use ***UtilParamsGet*** and ***UtilParamsSet*** utility messages specified in section 6.1.3 to read and modify parameters for tape devices. The list of parameter groups and their format for Tape Storage class devices is specified in the following tables.

Table 6-27. Group 0000h - Tape Storage Device Information Parameter Group

GroupNumber	0000h
Group Type	SCALAR
Name	***DEVICE INFORMATION***
Description	Information that describes a tape storage device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	1 byte	DeviceType	Refer to Table 6-28. Tape Storage Device Types.
1	r	1 byte	NumberOfPaths	Number of access paths to the media. For support of dual/multi-ported devices.
2	r	2 bytes	PowerState	Operation set by the most recent Power Management message.
3	r	4 bytes	BlockSize	Block size (number of bytes). If medium is removable, report maximum supported size.
4	r	8 bytes	DeviceCapacity	Device capacity (number of bytes). If medium is removable, report maximum supported capacity.
5	r	4 bytes	DeviceCapabilitySupport	Device capabilities describes attributes of the device and are described in Table 6-29
6	r	4 bytes	DeviceState	State of the device. See Table 6-30.
7	r/w	4 bytes	VariableMode	Set to a value of 1 if device in variable block mode. Cleared if in fixed block mode.
8	r	4 bytes	WriteDensity	Device specific write density for the current medium. A value of zero indicates either selectable densities are not supported, or a legal device-specific density code.

To obtain values for fields in the Device Information group, the DDM can take any action necessary to satisfy the request, including repositioning the media.

Table 6-28. Tape Storage Device Types

DeviceType	Description
00h	1/2” reel-to-reel
01h	QIC
02h	3480 form factor
03h	4mm
04h	8mm
05h	DLT
06h	Other
other values	reserved

Table 6-29. Device Capabilities

Capability	Description
bit 0	Caching – the device supports some form of caching
bit 1	Multi-path accessible – the device is accessible via multiple paths
bit 2	Supports compression
bit 3	Media removable
bit 4	Media Lockable
bit 5	Supports software load/unload operations
bit 6	Supports variable record sizes
bit 7	Supports fixed block record sizes
bit 8	Device is variable-partition capable
bit 9	Device is fixed-partition capable
bit 10	Device can predict device failures
bit 11	Device can predict media failures
bit 12	Device can informing when a head cleaning operation is required
bit 13	Device is environmental-warning capable
bit 14	Data Security
bit 15	Supports Tape Eject command

Table 6-30. Tape Storage Device State

Device State	Description
bit 0	Media present – media are present in the device
bit 1	PoweredOn – device is powered up
bit 2	Current medium is write protected
bit 3	Variable block is currently selected
bit 4	Compression is enabled for subsequent writes
bit 5	Current tape has compressed data for read operations
bit 6	Media locked
bit 7	Device failure predicted
bit 8	Media failure predicted
bit 9	Head cleaning operation is required
bit 10	Device environmental warning
bit 11	Caching
bit 12	Data Security is enabled for subsequent writes
bit 13	Current tape has Data Security data for read operations

Table 6-31. Group 0001h - Tape Storage Operational Control Parameter Group

GroupNumber	0001h
Group Type	SCALAR
Name	***OPERATIONAL CONTROL***
Description	Operational control parameters for a tape storage device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r/w	1 byte	RetryAttempts	Number of times a DDM/Device will retry a request before failing.
1	r	1 byte	reserved1	
2	r	2 bytes	reserved2	
3	r/w	4 bytes	RWTimeout	Read/Write timeout increment per block transfer in microseconds
4	r/w	4 bytes	LongPositionTimeout	Position timeout in microseconds for longest position operation.
5	r/w	4 bytes	ShortPositionTimeout	Position timeout in microseconds for short position operation, like a backhitch.
6	r/w	4 bytes	EraseTimeout	Erase timeout in microseconds.
7	r/w	4 bytes	TimeoutBase	Base Timeout for standard operations. The timeout value is subject to a multiplier determined by the message type in microseconds.

Table 6-32. Group 0002h - Tape Storage Power Control Parameter Group

GroupNumber	0002h
Group Type	SCALAR
Name	***POWER CONTROL***
Description	Configures how the tape storage device responds during inactivity and recovery from a powered down state.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r/w	4 bytes	PowerdownTimeout	The device powers down if it is not accessed within the allotted time (in microseconds). A zero value indicates the device will never power down.
1	r/w	4 bytes	OnAccess	Determines what the device does when accessed in a powered down state. Any block storage class request against the media constitutes access. bit 0: PowerUpOnAccess bit 1: LoadOnAccess

Table 6-33. Group 0003h - Tape Storage Cache Control Parameter Group

GroupNumber	0003h
Group Type	SCALAR
Name	***CACHE CONTROL***
Description	Information and control parameters for the cache of a tape storage device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description	
0	r	4 bytes	TotalCacheSize	Total available cache (in bytes).	
1	r/w	4 bytes	ReadCacheSize	Total available cache size for reads (in bytes).	
2	r/w	4 bytes	WriteCacheSize	Total available cache size for writes (in bytes).	
3	r/w	1 byte	WritePolicy	Policy employed by the cache when handling write requests.	
				00h	None/Disabled
				01h	WriteToCache
				02h	WriteThruCache
4	r/w	1 byte	ReadPolicy	Policy employed by the cache when handling read requests.	
				00h	None/Disabled
				01h	ReadCache
				02h	ReadAheadCache
				03h	ReadReadAheadCache
5	r	1 byte	ErrorCorrection	Error correction scheme.	
				00h	None/Disabled
				01h	Unknown
				02h	Other
				03h	Parity
				04h	SingleBitECC
				05h	MultiBitECC

Table 6-34. Group 0004h - Tape Storage Media information Parameter Group

GroupNumber	0004h
Group Type	SCALAR
Name	***MEDIA INFORMATION***
Description	

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	Capacity	Formatted capacity (in bytes) for current medium.
1	r	4 bytes	BlockSize	Block size (in bytes) for current medium.

Table 6-35. Group 0005h - Tape Storage Error Log Parameter Group

GroupNumber	0005h
Group Type	TABLE
Name	***ERROR LOG***
Description	Table of information for each error encountered. The client uses this information for its system log.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 bytes	ErrorDataIndex	Unique index that identifies each error log entry.
1	r	1 byte	Function	The function code of the failed request
2	r	1 byte	RetryCount	The number of times the function was unsuccessfully tried
3	r	2 bytes	ErrorStatusCode	Status Code describing the error or failure as defined in Table 6-20. The most significant four bits contain the TapePos code defined in Table 6-19.
4	r	2 bytes	reserved2	
5	r	4 bytes	TransferCount	The number of bytes (in variable mode) or records (in fixed block mode) successfully transferred before error occurred
6	r	8 bytes	TimeStamp	Number of microseconds from some fixed reference. The time interval between any two events can be found by the difference between their time stamps
7	r	8 bytes	UserInfo	Additional user information. The structure of this data varies for different types of access (e.g., SCSI, ATA) and is well known in the industry.

Table 6-36. Group 0100h - Tape Storage Historical Statistics Parameter Group

GroupNumber	0100
Group Type	SCALAR
Name	***STORAGE HISTORICAL STATS***
Description	

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 bytes	DevPowerOnHours	Approximate total power-on hours for the device (zero, if unknown or unsupported)
1	r	4 bytes	DevTapeMotionHours	Approximate total tape motion hours for the device (zero, if unknown or unsupported)
2	r	4 bytes	MediaWriteRetries	Historical count of write retries (device dependent) for the current medium (zero, if unknown or unsupported)
3	r	4 bytes	MediaReadRetries	Historical count of read retries (device dependent) for the current medium (zero if unknown or unsupported)
4	r	4 bytes	MediaLoadCount	Number of times the current medium has been loaded onto a device (zero if unknown or unsupported)

Table 6-37. Group 0101h - Tape Storage Runtime Statistics Parameter Group

GroupNumber	0101
Group Type	SCALAR
Name	***STORAGE RUNTIME STATS***
Description	

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 bytes	MediaWriteRetries	Current count of write retries (device dependent) for the entire current medium (zero if unknown or unsupported)
1	r	4 bytes	MediaReadRetries	Current count of read retries (device dependent) for the entire current medium (zero if unknown or unsupported)
2	r	4 bytes	CompressionRate	Compression rate for the current partition of the current medium, multiplied by 1000 (zero if unknown or unsupported)

Table 6-38. Group 0102h - Tape Storage Flexible Statistics Parameter Group

GroupNumber	0102
Group Type	SCALAR
Name	***STORAGE FLEXIBLE STATS***
Description	

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0				tbs.

6.6 SCSI Peripheral Class

The DDM for a SCSI port must support two classes of messages: one manages the SCSI controller and the second abstracts access to peripherals residing on the SCSI bus. This section defines the latter.

6.6.1 Overview

The SCSI peripheral class driver exposes an interface to the individual peripheral devices on a SCSI bus. The interface allows direct manipulation of the individual devices, but does not give access to the SCSI bus itself. This separation between the SCSI bus and the peripheral devices on the SCSI bus lets an operating system grant user-level applications access to devices on the SCSI bus, without possibly interfering with other devices or the SCSI bus itself.

SCSI peripheral class drivers are typically part of a SCSI adapter class driver. A single DDM exposes both interfaces, but separate TIDs are assigned to the SCSI adapter and to each peripheral device on the SCSI bus. The DDM is not responsible for policing the interfaces beyond securing the separation between the various TIDs, where requests to a particular TargetAddress (device) do not affect other devices. However, resetting the host bus adapter or the SCSI bus may dramatically affect the operations of devices on the bus. The operating system must restrict access to the SCSI Adapter class drivers to trusted applications only.

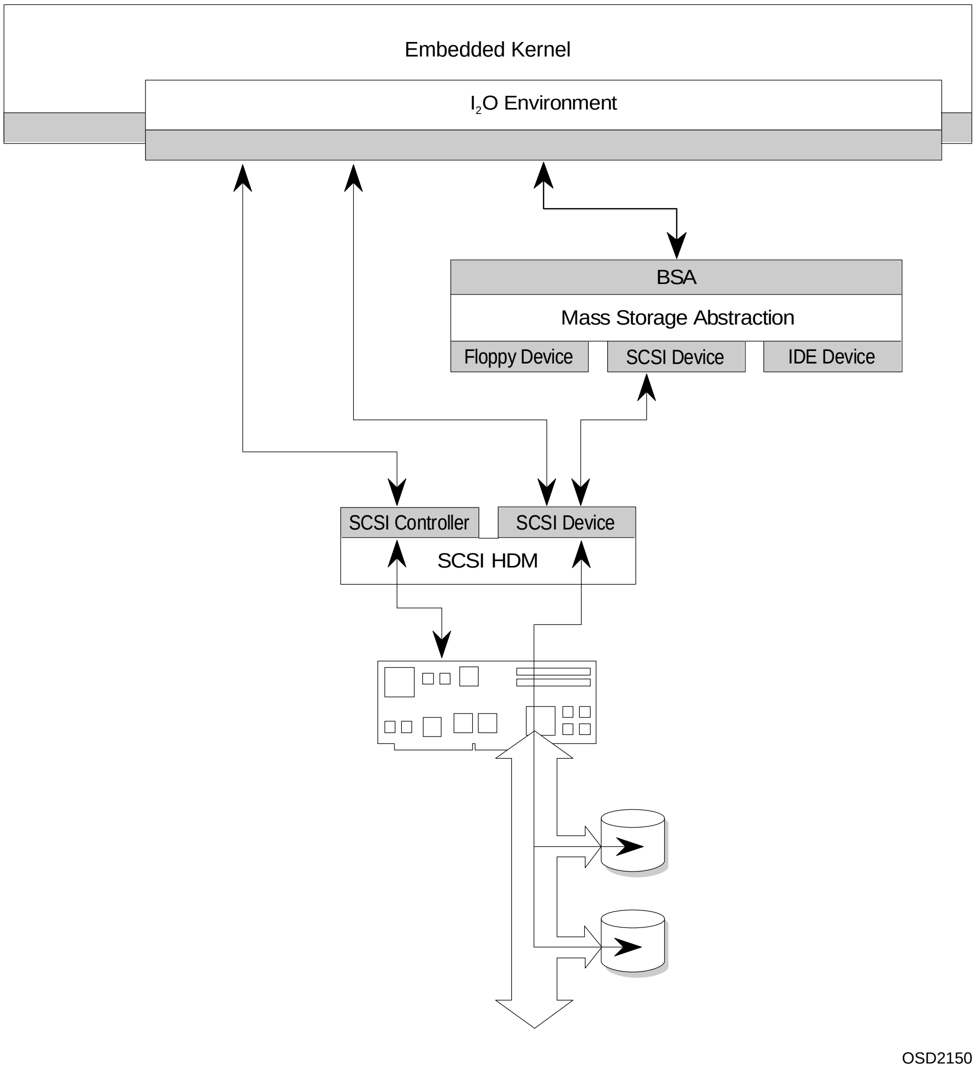


Figure 6-60. SCSI DDM Example

6.6.2 SCSI Reply Messages

Note: The DDM never sets the FAIL bit in the MessageFlags field. If the DDM receives a request with an unknown Function code or an ill-formed message, it replies with a ***Transaction Error Reply Message*** as specified in Chapter 3.

Replies to SCSI Peripheral requests come in two types: completion status replies and progress replies.

A completion status reply is generated for every request message. The ReqStatus field conveys the first-level completion status. For requests that completed without errors, only the ReqStatus field is set to reflect successful completion, the DetailedStatusCode is set to zero, the TransferCount field indicates the actual amount of data transferred, and there is no StatusData.

<table>
<tr><td colspan="2">3</td><td colspan="2">2</td><td colspan="2">1</td><td colspan="2">0</td><td></td></tr>
<tr><td>31</td><td>24</td><td>23</td><td>16</td><td>15</td><td>8</td><td>7</td><td>0</td><td></td></tr>
<tr><td colspan="4">MessageSize</td><td colspan="2">1 1 0 0 0 0 x 0</td><td colspan="2">VersionOffset = 01h</td><td>0</td></tr>
<tr><td colspan="2">Function</td><td colspan="3">InitiatorAddress</td><td colspan="3">TargetAddress</td><td>4</td></tr>
<tr><td colspan="8">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="8">TransactionContext</td><td>12 (16)</td></tr>
<tr><td colspan="2">ReqStatus</td><td colspan="2">reserved</td><td colspan="4">DetailedStatusCode</td><td>16 (24)</td></tr>
<tr><td colspan="8">TransferCount</td><td>20 (28)</td></tr>
<tr><td colspan="8">Status Data</td><td></td></tr>
</table>

offset values for 64-bit context field size in ()

Figure 6-61. Reply Message Template for SCSI Peripheral Class

The DDM always sets the TransferCount field to the actual number of bytes transferred. If the request did not require data transfer, it sets the value to zero.

The DetailedStatusCode is divided into two eight-bit fields, DeviceStatus and AdapterStatus. The DDM returns the SCSI device completion status associated with the request in the DeviceStatus field, per Table 6-39. It also returns status of the adapter in the AdapterStatus field, per Table 6-40. These codes are based on the CAM-1 definitions, but there are exceptions. First, zero is defined as *successful completion*. This is in contrast to CAM-1, which defines zero as *operation in progress*. Second, the value of one is permanently reserved. This value CAM-1 defines as *successful completion*. Finally, the presence of sense data is determined from the AutoSenseTransferCount field. CAM-1 uses a bit flag in the adapter status field to indicate the presence of sense data.

For requests that did not complete successfully, a detailed error code is returned in the DetailedStatusCode fields. If the AutoSense flag is set in the ScbFlags of the request, the DDM executes a ***request sense*** command and returns sense data as specified. The DDM indicates the amount of sense data returned in the AutoSenseTransferCount field. The request either provides a reply buffer for the sense data or the DDM returns it as part of the reply payload. Up to 40 bytes of sense data may be returned in the reply frame. If a sense buffer is specified, then the sense data is written to the sense buffer and is excluded from the reply frame. See the SCSI control block flags in section 6.6.4.2.

3	2	1	0	
31 … 24	23 … 16	15 … 8	7 … 0	
MessageSize		1 1 0 0 0 0 x 0	VersionOffset = 01h	0
Function	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
ReqStatus	reserved	AdapterStatus	DeviceStatus	16 (24)
Transfer_Count				20 (28)
AutoSenseTransferCount				24 (32)
SenseData				28 (36)
...				
(up to 40 bytes)				60

offset values for 64-bit context field size in ()

Figure 6-62. Unsuccessful Completion Reply Message for SCSI Peripheral Class

The DetailedStatusCode field contains as specific a code as possible to describe the failure.

Table 6-39. SCSI Device Completion Status Codes

DeviceStatus	Description
SCSI_SUCCESS	Success – no warnings
SCSI_CHECK CONDITION	Check condition
SCSI_BUSY	Busy
SCSI_RESERVATION_CONFLICT	Reservation conflict
SCSI_COMMAND_TERMINATED	Command terminated
SCSI_TASK_SET_FULL	Task Set Full
SCSI_ACA_ACTIVE	ACA Active

Table 6-40. SCSI Adapter Status Codes for Device Operations

AdapterStatus	Description
HBA_SUCCESS	Success – no warnings
HBA_ADAPTER_BUSY	Cannot process request
HBA_AUTOSENSE_FAILED	AutoSense operation failed
HBA_BDR_MESSAGE_SENT	SCSI Bus Device Reset message sent
HBA_CDB_RECEIVED	SCSI CDB has been received
HBA_COMMAND_TIMEOUT	Timeout on request
HBA_COMPLETE_WITH_ERROR	Request completed with an error
HBA_DATA_OVERRUN	SCSI data phase overrun
HBA_DEVICE_NOT_PRESENT	SCSI device not present
HBA_FUNCTION_UNAVAILABLE	Requested function is not available
HBA_IDE_MESSAGE_SENT	SCSI Initiator Detected Error message sent
HBA_INVALID_CDB	Invalid SCSI CDB received in host target mode
HBA_LUN_ALREADY_ENABLED	SCSI LUN is already enabled
HBA_LUN_INVALID	LUN supplied is invalid
HBA_MESSAGE_RECEIVED	SCSI message received in host target mode
HBA_MR_MESSAGE_RECEIVED	SCSI Message Reject message received
HBA_NO_ADAPTER	No Host Bus Adapter detected
HBA_NO_NEXUS	SCSI nexus is not established
HBA_PARITY_ERROR_FAILURE	Uncorrectable SCSI bus parity error
HBA_PATH_INVALID	Path supplied is invalid
HBA_PROVIDE_FAILURE	Unable to provide required capability
HBA_QUEUE_FROZEN	Adapter queue frozen with this error
HBA_REQUEST_ABORTED	Request aborted by the host
HBA_REQUEST_INVALID	Request is invalid
HBA_REQUEST_LENGTH_ERROR	Request length supplied is inadequate
HBA_REQUEST_TERMINATED	Request terminated by the host
HBA_RESOURCE_UNAVAILABLE	Resource unavailable
HBA_SCSI_BUS_BUSY	SCSI bus is busy
HBA_SCSI_BUS_RESET	SCSI bus reset occurred
HBA_SCSI_IID_INVALID	SCSI initiator ID is invalid
HBA_SCSI_TID_INVALID	SCSI target device ID supplied is invalid
HBA_SELECTION_TIMEOUT	SCSI device did not respond to selection
HBA_SEQUENCE_FAILURE	SCSI bus phase sequence failure
HBA_UNABLE_TO_ABORT	Cannot abort request
HBA_UNABLE_TO_TERMINATE	Cannot terminate request
HBA_UNACKNOWLEDGED_EVENT	Unacknowledged event by host
HBA_UNEXPECTED_BUS_FREE	Unexpected SCSI bus free

6.6.3 Support for Utility Messages

6.6.3.1 Events

The SCSI peripheral driver supports the generic events specified in section 6.1.3.8, Table 6-4 for ***UtilEventRegister***. In addition, the following SCSI events are specified.

Table 6-41. SCSI EventIndicator Assignments

Event Name	Bit	Description
SCSI_SMART	4	Reports SCSI SMART data indication

Table 6-42. EventData for SCSI Events

Event Name	Event Data
SCSI_SMART	SCSI ASC and ASCQ (2 bytes)

6.6.3.2 Getting and Setting Parameters

Both the client (service user) and management use the ***UtilParamsGet*** and ***UtilParamsSet*** utility messages to read and modify parameters for the SCSI Peripheral class devices. Refer to section 6.1.3, *Utility Messages*. The list of parameter groups for SCSI Peripheral class devices is specified in Table 6-45. The ***UtilParamsGet*** utility request (Group = *DEVICE_INFORMATION*) causes the DDM to return the current operating parameters associated with the device.

6.6.4 SCSI Peripheral Request Messages

Table 6-43 shows requests that can be made to SCSI Peripheral class objects.

Table 6-43. Request Messages for the SCSI Peripheral Class

Function	Description
ScsiDeviceReset	Reset the target device
ScsiScbAbort	Terminate the SCB
ScsiScbExec	Execute the SCB on the target unit

All SCSI Peripheral class messages are single-transaction messages. Typically, the MessageFlags field for requests should be set to 00h (for 32-bit context size) or 02h (for 64-bit context size). For a normal completion reply message, the MessageFlags field should contain C0h (for 32-bit context size) or C2h (for 64-bit context size). Since some requests provide an SGL, the value of the VersionOffset field depends on the location of the SGL. Since all replies are single-transaction replies, the VersionOffset field should be set to 01h for all replies.

6.6.4.1 Device Reset

The ***ScsiDeviceReset*** request causes the DDM to soft reset the SCSI device associated with the TargetAddress (e.g., issue a bus device reset command). All requests currently queued at the DDM for execution are aborted and returned immediately with an *ABORTED* status before the reset command is complete.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
ScsiDeviceReset	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)

offset values for 64-bit context field size in ()

Figure 6-63. *ScsiDeviceReset* Request Message

6.6.4.2 SCSI Control Block Abort

The ***ScsiScbAbort*** request causes the DDM to remove the SCB from the execution queue if the SCB has not yet been sent to the device. If the SCB is already sent, the DDM attempts abort the SCB at the device. If the DDM removes the SCB from its queue or aborts its execution at the device, it returns a *STATUS_CODE_SUCCESS* completion code for this request and an *aborted* completion message for the aborted request.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
ScsiScbAbort	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
TransactionContextToAbort				20 (24)

offset values for 64-bit context field size in ()

Figure 6-64. ScsiScbAbort Request Message

6.6.4.3 SCSI Control Block Execute

The ***ScsiScbExec*** request causes the DDM to send the accompanying CDB to the device associated with the TargetAddress for execution.

If no data transfer will take place, the ByteCount and SGL are not present and the total length of this request's message size is reduced to 36 bytes.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
ScsiScbExec	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
ScbFlags		Reserved	CdbLen	16 (24)
CDB				20 (28)
...				24 (32)
...				28 (36)
(16 Bytes)				32 (40)
ByteCount				36 (44)
SGL				40 (48)

offset values for 64-bit context field size in ()

Figure 6-65. *ScsiScbExec* Request Message

Fields

VersionOffset A value of 0A1h for 32-bit context size and C1h for 64-bit context size.

CdbLen Number of actual bytes of CDB in CDB field.

ScbFlags The ScbFlags impose additional requirements on the command execution phase, as defined in Table 6-44. These requirements are for the duration of this SCB only.

Table 6-44. ScbFlags Bit Definitions

Bits:			Definition:
Bit 15	**Bit 14**		**TransferDirection**
0	0		*NO_DATA_TRANSFER*
0	1		*FROM_DEVICE_TO_SYSTEM*
1	0		*FROM_SYSTEM_TO_DEVICE*
1	1		Reserved
Bit 13:			**DisconnectMode**
0			*DISABLE_DISCONNECT*
1			*ENABLE_DISCONNECT*
Bits 12-10:			**reserved bits**
Bit 9	**Bit 8**	**Bit 7**	**TagType**
0	0	0	*NO_TAG_QUEUEING*
0	0	1	*SIMPLE_TAG*
0	1	0	*HEAD_OF_QUEUE_TAG*
0	1	1	*ORDERED_QUEUE_TAG*
1	0	0	*ACA_QUEUE_TAG*
Bit 6	**Bit 5**		**AutoRequestSense**
0	0		*DISABLE_AUTOSENSE*
0	1		*RETURN_SENSE_DATA_IN_REPLY_MESSAGE_FRAME*
1	0		Reserved
1	1		*RETURN_SENSE_DATA_IN_REPLY_BUFFER*
Bits 4-0:			**reserved bits**

When the TagType field is set to *NO_TAGGED_COMMAND_QUEUING*, it overrides the current operating parameters and sends the command without tagging it. To accomplish this, the DDM may have to wait for outstanding tagged commands to complete.

If the AutoRequestSense is not disabled, the DDM executes a request sense command as required. The sense data is either returned in the reply frame, or in a separate sense buffer, as specified. If the ScbFlags indicates *RETURN_SENSE_DATA_IN_REPLY_MESSAGE_FRAME*, up to 40 bytes of sense data can be returned. In this case, the SGL does not specify a request sense buffer. The AutoSenseTransferCount field contains the actual size of sense data returned.

If the ScbFlags indicates *RETURN_SENSE_DATA_IN_REPLY_BUFFER*, then the SGL specifies the sense buffer. In this case, no sense data is returned in the reply frame. The sense data buffer is always second in the SGL. The first buffer contains the data transfer associated with execution of the SCSI CDB. If the SCSI CDB requires a sense buffer, but not a data transfer, then a zero-length buffer must be the first buffer.

6.6.5 Managing SCSI Peripheral Parameters

Both the client (service user) and management use ***UtilParamsGet*** and ***UtilParamsSet*** utility messages in section 6.1.3 to read and modify parameter for SCSI peripheral. The list of parameter groups and their format for SCSI peripheral class devices is specified in the following tables.

Table 6-45. SCSI Peripheral Parameter Groups

GroupNumber	0000h
GroupType	SCALAR
Name	***DEVICE_INFORMATION***
Description	Describes the capabilities of the SCSI peripheral device.

FieldIdx	(r/w)	Field Size	Parameter	Description
0	r	1 byte	DeviceType	SCSI Device Types 00h Direct-access read/write storage device (e.g., magnetic disk) 01h Sequential-access storage device (e.g., magnetic tape) 02h Printer device 03h Processor device 04h Write-one storage device (e.g., WORM disk) 05h CD-ROM device 06h Scanner device 07h Optical Memory device (e.g., MO disks) 08h Medium Changer device (e.g.,jukeboxes) 09h Communications device 0Ah, 0Bh Defined by ASC IT8 (graphic arts pre-press device) 0Ch Array controller device (e.g., RAID array controller) 0Dh-1Eh Reserved 1Fh Unknown or no device type 20h-FFh Reserved **Note:** *See SCSI-2 or SCSI-3 for further details.*
1	r/w	1 byte	Flags	Bit 0: SCSI Peripheral Type 0 - Parallel 1 - Serial Bit 1 : Reserved Bit 2 : Disconnect/Reconnect 0 - Disable Disconnect/Reconnect 1 - Enable Disconnect/Reconnect Bits 3 & 4: MODE 00 = Set Parameters from Message Payload 01 = Ignore Payload and Set Parameters to default 10 = Ignore Payload and Set Parameters to Safest possible 11 = reserved Bits 5 & 6 : MAXIMUM DATA WIDTH 00 = 8 bits 01 = 16 bits 10 = 32 bits 11 = Reserved Bit 7: AcknowledgmentSynchronous Negotiations 0 = Do not initiate acknowledgmentsynchronous negotiations 1 = Initiate acknowledgmentsynchronous negotiations

GroupNumber	0000h
GroupType	SCALAR
Name	***DEVICE_INFORMATION***
Description	Describes the capabilities of the SCSI peripheral device.

FieldIdx	(r/w)	Field Size	Parameter	Description
2	r	2 bytes	Reserved2	
3	r	4 bytes	Identifier	The SCSI target/initiator ID for this device
4	r	8 bytes	LUN	Logical unit information, per the SCSI-3 Controller Commands Standard. SCSI-2 compatible LUNs go into byte offset 1.
5	r/w	4 bytes	QueueDepth	Maximum number of CDBs that can be sent to the device. To find the largest number supported by this device, issue a ***UtilParamSet*** message with this field set to -1, then issue a ***UtilParamGet*** message to read the largest value supported. The DDM adjusts the value to the largest the device can support, if it can be determined. Otherwise, the value of -1 is returned. The number of CDBs issued to the device can be limited by some other resource. Sending a Queue Depth value of 0 (zero) causes the DDM to choose a default value. Setting the Queue Depth to 1 (one) causes the DDM to disable tagged command queuing.
6	r	1 byte	Reserved1	
7	r/w	1 byte	NegOffset	The negotiated synchronous offset for this device. A ***UtilParamGet*** value of 0 indicates the transfer mode is asynchronous.
8	r/w	1 byte	NegDataWidth	Negotiated data width (in bits).
9	r	1 byte	Reserved1	
10	r	8 bytes	NegSyncRate	Reading this value returns the current negotiated transfer rate, in kilo-transfers per seconds, for this device. A value of 0 shall represent asynchronous mode. Setting the value to -1 indicates that the transfer rate should be set to the maximum possible..

GroupNumber	0001
GroupType	SCALAR
Name	***BUS PORT INFORMATION***
Description	Describes the bus port of the SCSI peripheral device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description

GroupNumber	0001
GroupType	SCALAR
Name	***BUS PORT INFORMATION***
Description	Describes the bus port of the SCSI peripheral device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	1 byte	Physical/Electrical	Physical and/or electrical characteristics of the bus port. 01h Other 02h Unknown 03h SCSI Parallel Interface 04h SCSIFibre Channel Protocol 05h SCSI Serial Bus Protocol (1394) 06h SCSI Serial Storage Architecture
1	r	1 byte	Electrical Interface	Electrical interface characteristics of the bus port. 01h Other 02h Unknown 03h Single Ended 04h Differential 05h Low Voltage Differential 06h Optical
2	r	1 byte	Isochronous	Indicates if the bus port supports isochronous transfers. 00h Does not support isochronous transfers 01h Supports isochronous transfers 02h Unknown
3	r	1 byte	ConnectorType	Physical connection for this bus port. 01h Other 02h Unknown 03h None 04hS CSI (A) High-Density Shielded (50 pins) 05h SCSI (A) High-Density Unshielded (50 pins) 06h SCSI (A) Low-Density Shielded (50 pins) 07h SCSI (A) Low-Density Unshielded (50 pins) 08h SCSI (P) High-Density Shielded (68 pins) 09h SCSI (P) High-Density Unshielded (68 pins) 0Ah SCSI SCA-I (80 pins) 0Bh SCSI SCA-II (80 pins) 0Ch SCSI Fibre Channel DB9 (Copper) 0Dh SCSI Fibre Channel (fibre) 0Eh SCSI Fibre Channel SCA-II (40 Pins) 0Fh SCSI Fibre Channel SCA-II (20 Pins) 10hSCSI Fibre Channel BNC
4	r	1 byte	ConnectorGender	Indicates the gender of the connector. 01h Other 02h Unknown 03h Female 04h Male
5	r	1 byte	Reserved1	
6	r	2 bytes	Reserved2	

GroupNumber	0001
GroupType	SCALAR
Name	***BUS PORT INFORMATION***
Description	Describes the bus port of the SCSI peripheral device.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
7	r	4 bytes	MaxNumberDevices	Maximum directly addressable entities supported by bus ports protocol.

6.7 Bus Adapter Class

When an adapter contains one or more ports attaching some number of peripheral devices, the DDM abstracts each peripheral as an individual device. This abstraction provides control for each individual device independent from the others. A number of bus characteristics and functions affect all of the peripherals concurrently, and thus can not be abstracted by any single device. Therefore the *Bus Adapter Class* provides for the management and control of the secondary bus independent of the underlying technology.

An example is a SCSI adapter that provides access to a SCSI bus (SCSI channel) attaching a number of SCSI devices. The SCSI Peripheral class provides the abstraction of each SCSI peripheral device. The DDM for the adapter registers a SCSI peripheral device for each peripheral device on the bus. In addition, the DDM registers a bus adapter device for the SCSI bus controller (adapter).

This section defines the messages and behavior of such bus adapter devices.

6.7.1 Overview

The Host Bus Adapter (HBA) class device exposes an interface for controlling the bus adapter and its secondary bus. Access to this class driver is assumed to be under the control of the OS and its security policies. This means that the primary user is the OS or one of its agents. Other facilities needing the services provided by the HBA device must claim as secondary or authorized users.

For multi-channel adapters, the DDM registers each secondary bus as a HBA class device and thus associates each bus with a different TargetAddress. Some operations, such as resetting an adapter entirely, can affect all secondary buses on that adapter.

6.7.2 Reply Messages

The DDM generates a completion status reply for every received request message. The ReqStatus field conveys the first-level completion status. The DetailedStatusCode is divided into two eight-bit fields; AdapterStatus and a reserved field. The DDM sets the reserved field to zero and returns the status of the adapter in the AdapterStatus field, per Table 6-46.

For requests that complete successfully, the DDM sets the ReqStatus field to reflect successful completion and sets the DetailedStatusCode to zero or a code that identifies conditions that were overcome in completing the request.

For requests that do not complete successfully, the DDM returns an error code in the AdapterStatus field.

The ReplyPayload field is reserved for future definition. The DDM must set MessageSize to indicate no reply payload.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		1 1 0 0 0 0 x 0	VersionOffset = 01h	0
Function	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransactionContext				12 (16)
ReqStatus	reserved	AdapterStatus	reserved	16 (24)
ReplyPayload				20 (28)

offset values for 64-bit context field size in ()

Figure 6-66. Reply Message Structure for Bus Adapter Class

Note: The DDM never sets the FAIL bit in the MessageFlags field. If the DDM receives a request with an unknown Function code or an ill-formed message, it replies with a ***Transaction Error Reply Message*** as specified in Chapter 3.

The AdapterStatus field contains as specific a code as possible to describe the failure.

Table 6-46. AdapterStatus Values (part of DetailedStatusCode)

AdapterStatus	Description
HBA_SUCCESS	Success – no warnings
HBA_ADAPTER_BUSY	Cannot process request
HBA_COMMAND_TIMEOUT	Timeout on request
HBA_COMPLETE_WITH_ERROR	Request completed with an error
HBA_FUNCTION_UNAVAILABLE	Requested function is not available
HBA_NO_ADAPTER	No Host Bus Adapter detected
HBA_PARITY_ERROR_FAILURE	Uncorrectable bus parity error
HBA_PATH_INVALID	Path supplied is invalid
HBA_PROVIDE_FAILURE	Unable to provide required capability
HBA_QUEUE_FROZEN	Adapter queue frozen with this error
HBA_REQUEST_ABORTED	Request aborted by initiator
HBA_REQUEST_INVALID	Request is invalid
HBA_REQUEST_LENGTH_ERROR	Request length supplied is inadequate
HBA_REQUEST_TERMINATED	Request terminated by initiator
HBA_RESOURCE_UNAVAILABLE	Resource unavailable
HBA_BUS_BUSY	Secondary bus is busy
HBA_BUS_RESET	Secondary bus reset occurred
HBA_ID_INVALID	Secondary target device ID supplied is invalid
HBA_SEQUENCE_FAILURE	Secondary bus phase sequence failure
HBA_UNABLE_TO_ABORT	Cannot abort request
HBA_UNABLE_TO_TERMINATE	Cannot terminate request
HBA_UNACKNOWLEDGED_EVENT	Unacknowledged event by user
HBA_UNEXPECTED_BUS_FREE	Unexpected secondary bus release

6.7.3 Support for Utility Messages

6.7.3.1 Events

The bus adapter supports the generic events specified in section 6.1.3.8, Table 6-1 for ***UtilEventRegister***. No additional events are defined.

6.7.3.2 Getting and Setting Parameters

Both the client (service user) and management use the ***UtilParamsGet*** and ***UtilParamsSet*** utility messages to read and modify parameters for the bus adapter class devices. Refer to section 6.1.3, *Utility Messages*. The list of parameter groups for bus adapter class devices is in Table 6-48. The ***UtilParamsGet*** utility request (Group = 0000h) causes the DDM to return the operating parameters associated with the bus.

6.7.4 Bus Adapter Class Request Messages

Table 6-47 shows base class requests that can be made to Bus Adapter class devices.

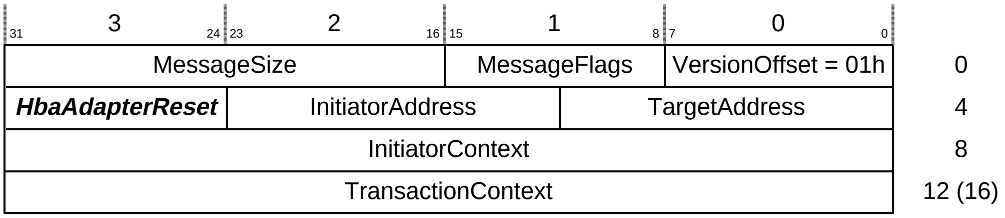
Table 6-47. Request Messages for the Bus Adapter Class

Function	Description
HbaAdapterReset	Reset the Adapter
HbaBusQuiesce	Suspends I/O to all devices on the secondary bus
HbaBusReset	Reset the secondary bus
HbaBusScan	Scan the secondary bus

All Bus Adapter class messages are single-transaction messages. Typically, the MessageFlags field for requests should be set to 00h (for 32-bit context size) or 02h (for 64-bit context size). For a normal reply, it should contain C0h (for 32-bit context size) or C2h (for 64-bit context size). Since no request provides a SGL, and all replies are single-transaction, the VersionOffset field should be set to 01h for both requests and replies.

6.7.4.1 Reset Host Bus Adapter

The ***HbaAdapterReset*** request causes the DDM to hard reset the bus adapter. Resetting the bus adapter does not always reset the bus itself. When an adapter is reset, the DDM aborts all outstanding requests to all devices on any buses connected to the adapter. All aborted requests return with the *ABORTED* status before the ***HbaAdapterReset*** request completes.



31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
HbaAdapterReset	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)

offset values for 64-bit context field size in ()

Figure 6-67. *HbaAdapterReset* Request Message

6.7.4.2 Bus Quiesce

This message enables and disables (quiesces) secondary bus activity. An odd value in the Flags field means quiesce the bus. An even value in this field means resume normal bus function. While quiesced, the adapter does not source commands or data on the secondary bus.

The behavior of a ***HbaBusQuiesce*** shall be as follows:

1. The HDM receives the ***HbaBusQuiesce*** request with Flags = ***quiesce***.
2. If the primary user registered for StateSensitive (see ***UtilClaim*** message) and registered for *I2O_EVENT_IND_ STATE_CHANGE* (see ***UtilEventRegister*** message), then the DDM sends the primary user an ***UtilEventRegister*** reply and waits for the ***UtilEventAck*** message. If the DDM receives a ***HbaBusQuiesce*** request with Flags = ***quiesce*** before it receives the ***UtilEventAck*** message, then it rejects the original ***HbaBusQuiesce*** request. Otherwise the DDM accepts the ***HbaBusQuiesce*** request.
3. For any peripheral class messages processed after the HDM accepts the ***HbaBusQuiesce***, the HDM either:

a) immediately generates a busy status (***TransactionError*** reply message with *I2O_Detail_Status_Device_Busy* detailed status code) reply, **OR**

b) queues the request until full, then it issues the ‘device busy’ reply to subsequent requests**.**

4. The HDM waits for all outstanding commands to complete, returning the appropriate reply.
5. The HDM returns complete to the original ***HbaBusQuiesce*** request.
6. The HDM generates the I2O_EVENT_IND_STATE_CHANGE (StateChange=SUSPENDED)event for each peripheral served by the bus.
7. For base class messages sent to any of those peripheral devices, the HDM continues to queue commands or issue the ‘adapter busy’ reply until it receives a ***HbaBusQuiesce*** request with Flags = ***normal***. (**<u>Note</u>**: the HDM must have queue space for ***HbaBusQuiesce*** to resume normal bus operation. Thus when queue depth reaches a value of <u>maximum -1</u>, the HDM must return ‘adapter busy status.)
8. When the HDM receives the ***HbaBusQuiesce*** request with Flags = ***normal***, the HDM resumes normal operation and the HDM generates an I2O_EVENT_IND_STATE_CHANGE event for each peripheral indicating ‘return to normal operation’.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
HbaBusQuiesce	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)
Flags				16 (24)

offset values for 64 bit context field size in ()

Figure 6-68. *HbaBusQuiesce* Request Message

Gross timeouts at the ISM and OSM level should be resolved by hints provided by the HDM at initialization time.

6.7.4.3 Bus Reset

The ***HbaBusReset*** request tells the DDM to perform a hard reset on the secondary bus associated with TargetAddress. For a multi-channel adapter, resetting one channel should not affect the operation of the other channel. However, if the design of the adapter prevents separating the channels, all outstanding requests on all channels should be aborted with an *ABORTED* status.

If the bus reset causes the peripheral devices to be reset, then each peripheral device generates a state change event (StateChange=Reset/restarted).

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
HbaBusReset	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)

offset values for 64-bit context field size in ()

Figure 6-69. *HbaBusReset* Request Message

6.7.4.4 Bus Scan

When it receives the ***HbaBusScan*** request, the DDM scans the secondary bus associated with the TargetAddress for additional peripherals and confirms previously registered peripherals. The BusType specific parameter tables provide bounds that limit the DDM's search.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset = 01h	0
HbaBusScan	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TransactionContext				12 (16)

offset values for 64-bit context field size in ()

Figure 6.70. *HbaBusScan* Request Message

6.7.5 Modifying Configuration and Operating Parameters

Both the client (service user) and management use the ***UtilParamsGet*** and ***UtilParamsSet*** utility messages to read and modify parameter for Bus Adapter devices. The list of parameter groups for the Bus Adapter class is specified in the following tables:

Table 6-48. Bus Adapter Parameter Groups

GroupNumber	0000h
GroupType	SCALAR
Name	***CONTROLLER_INFORMATION***
Description	Describes the capabilities of the bus adapter.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	1 byte	BusType	Indicates secondary bus type and its version. 00h generic bus 01h SCSI 10h Fibre Channel
1	r	1 byte	BusState	value as per Table 6-5 EventData for StateChange.
2	r	2 bytes	reserved2	
3	r	12 byte	BusName	ASCII sting

GroupNumber	0100
GroupType	SCALAR
Name	***HISTORICAL_STATS***
Description	Statistical information for the bus controller.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 bytes	TimeLastPowerUp	The amount of time (in seconds) since this controller was last powered on.
1	r	4 bytes	TimeLastReset	The amount of time (in seconds) since last bus reset.

GroupNumber	0200h
GroupType	SCALAR - conditional on BusType = SCSI
Name	***SCSI_CONTROLLER_INFORMATION***
Description	Describes the capabilities of the SCSI bus adapter.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	1 byte	ScsiType	Indicates secondary bus type and its version. 00h unknown 01h SCSI-1 02h SCSI-2 03h SCSI-3
1	r	1 byte	ProtectionManagement	Indicates whether or not the controller provides redundancy or protection against device failures. 00h Other 01h Unknown 02h Unprotected 03h Protected 04h Protected through SCC standard
2	r/w	1 byte	Settings	bit 0 Parity Checking 0 - Disabled 1 - Enabled bit 1 Scan Order 0 - Low-to-High 1 - High-to-Low bit 2 Initiator ID 0 - Initiator ID is default 1 - Initiator ID is specified bit 3 SCAM 0 - Disabled 1 - Enabled bit 4-6 Reserved bit 7 Controller Type 0 - Parallel 1 - Serial Note: When ControllerType=1, only Fields 0, 1, 2, 3 and 7 apply.
3	r	1 byte	Reserved1	
4	r/w	4 bytes	InitiatorID	
5	r/w	8 bytes	ScanLun0Only	Bit position of a set bit indicates the SCSI-parallel target ID for which only LUN 0 should be scanned.
6	r/w	2 bytes	DisableDevice	Bit position of a set bit indicates the SCSI-parallel target ID of the device to be disabled. (e.g. for a value of 0001h, Target ID 0 will not be scanned)
7	r	1 byte	MaxOffset	The maximum synchronous offset that this adapter can support.
8	r	1 byte	MaxDataWidth	Maximum data width supported by this adapter.
9	r	8 bytes	MaxSyncRate	Theoretical maximum transfer rate, in kilo-transfers-per-seconds, that this adapter can achieve.

GroupNumber	0201
GroupType	SCALAR - conditional on BusType = SCSI
Name	***SCSI_BUS_PORT_INFORMATION***
Description	Describes the capabilities of the SCSI bus controller.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	1 byte	PhysicalElectrical	Physical / electrical characteristics of the bus port. 01h Other 02h Unknown 03h SCSI Parallel Interface 04h SCSI Fibre Channel Protocol 05h SCSI Serial Bus Protocol (1394) 06h SCSI Serial Storage Architecture
1	r	1 byte	ElectricalInterface	Electrical interface characteristics of the bus port. 01h Other 02h Unknown 03h Single Ended 04h Differential 05h Low Voltage Differential 06h Optical
2	r	1 byte	Isochronous	Indicates if the bus port supports isochronous transfers. 00h Does not support isochronous transfers 01h Supports isochronous transfers 02h Unknown
3	r	1 byte	ConnectorType	Physical connection for this bus port. 01h Other 02h Unknown 03h None 04h SCSI (A) High-Density Shielded (50 pins) 05h SCSI (A) High-Density Unshielded (50 pins) 06h SCSI (A) Low-Density Shielded (50 pins) 07h SCSI (A) Low-Density Unshielded (50 pins) 08h SCSI (P) High-Density Shielded (68 pins) 09h SCSI (P) High-Density Unshielded (68 pins) 0Ah SCSI SCA-I (80 pins) 0Bh SCSI SCA-II (80 pins) 0Ch SCSI Fibre Channel DB9 (Copper) 0Dh SCSI Fibre Channel (fibre) 0Eh SCSI Fibre Channel SCA-II (40 Pins) 0Fh SCSI Fibre Channel SCA-II (20 Pins) 10h SCSI Fibre Channel BNC
4	r	1 byte	ConnectorGender	Indicates the gender of the connector. 01h Other 02h Unknown 03h Female 04h Male
5	r	1 byte	Reserved1	
6	r	2 bytes	Reserved2	
7	r	4 bytes	MaxNumberDevices	Maximum number of directly-addressable entities supported by bus port's protocol.

8	r/w	4 bytes	DeviceIdBegin	First device to include in search when scanning SCSI bus for devices.
9	r/w	4 bytes	DeviceIdEnd	Last device to include in search.
10	r/w	8 bytes	LunBegin	First LUN to include in search.
11	r/w	8 bytes	LunEnd	Last LUN to include in search.

GroupNumber 0300h

GroupType SCALAR - conditional on BusType = Fibre Channel

Name ***FCA_CONTROLLER_INFORMATION***

Description Describes the capabilities of the Fibre Channel bus adapter.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	1 byte	FcaType	Indicates secondary bus type and its version. 00h unknown 01h Fibre Channel AL
1			tbd	tbd

GroupNumber 0301

GroupType SCALAR - conditional on BusType = Fibre Channel

Name ***FCA_PORT_INFORMATION***

Description Describes the capabilities of the Fibre Channel port controller.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	tbd	tbd	tbd	tbd
1	tbd	tbd	tbd	tbd

6.8 IDE Adapter Class

Not defined at this time – to be supplied at a later date.

6.9 Floppy Adapter Class

Not defined at this time – to be supplied at a later date.

6.10 LAN Class

This section describes the I$_2$O LAN abstraction model for LAN adapters and their ports. LAN types in this class include: Ethernet/IEEE802.3, 100BaseVG, IEEE802.5 Token Ring, FDDI, and Fibre Channel as a network. Wide Area Networks such as ATM have their own class definition and are not covered by the LAN class. The I$_2$O LAN abstraction layer is the primary interface host operating systems use to access LAN ports.

A LAN port provides access to a network. The DDM registers a different LAN class device for each port it provides. The user, typically an OSM, performs LAN operations by sending requests to, and listening for replies from, a LAN class device. The term *device* applies to the abstract interface produced by the message class. The client sends messages to that device by indicating the TID for that device in the message's TargetAddress field. The term *adapter* refers to the physical device or facility the DDM controls.

Note: The DDM never sets the FAIL bit in the MessageFlags field. If the DDM receives a request with an unknown Function code or an ill-formed message, it replies with a ***Transaction Error Reply Message*** as specified in Chapter 3.

6.10.1 Overview

Both sending and receiving can be in bursts, using batches and buckets, respectively. An additional level of batching in the IOP's communication layer's handles replies on interrupts. The effect is cumulative.

This version of the specification lets a LAN device generate its own MAC headers on packet transmissions. For receiving packets, only basic address filtering is provided.

6.10.2 Sends

The user (e.g., the LAN OSM) sends packets using the ***LanPacketSend*** or the ***LanSduSend*** request. For the ***LanPacketSend,*** the user supplies the complete packet. However, for ***LanSduSend,*** the LAN device supplies the MAC header and the user supplies the rest of the packet. If a packet is small enough to fit in the payload of a request frame, the user may supply it by immediate data mode (i.e., the packet is contained in the SGL element). For larger packets, arbitrary lists of packets, each containing arbitrary lists of memory fragments, can be supplied by a single message using any SGL addressing mode.

The DDM adds padding as required to meet media-specific, minimum packet size. The DDM does not need to perform any other length checking on the packet. The user must generate valid packets that contain sufficient data for valid addressing, and do not exceed the declared maximum packet length, as specified in the ClassInfo structure in Parameter Group 0.{}

The LAN device indicates whether it can generate its own MAC headers. If it can, the user may pass packets to the LAN device that do not contain MAC headers using the ***LanSduSend*** request. Otherwise, the user passes packets that include MAC headers exactly as they will transmit on the medium, using the ***LanPacketSend*** request.

The LAN class uses batch replies. That is, a single reply may acknowledge multiple packet transmissions of multiple requests. All packets acknowledged by a single reply have the same Transmission status. Typically, the user is concerned less with the timeliness of the transmission status than with the return of its resources. Since errors should occur infrequently and all packets reported in the same reply must have the same status, the LAN device cannot batch send replies for packets with errors.

When a batch of packets is supplied to the DDM for transmission, it owns the memory they are in until transmission completes. Then, transmission status is reported and ownership returns to the user.

The user may reclaim packets posted for sending using the ***UtilAbort*** request.

6.10.2.1 Processing Send Replies

The LAN device returns results of send requests (i.e., ***LanPacketSend*** and ***LanSduSend*** messages) to the user by posting replies. Every reply contains an InitiatorContext that is an exact copy of the InitiatorContext value from some send request. Allocating the InitiatorContext field is OS-specific, with the assumption that the OS routes the reply to the LAN user based on this field. In addition, each packet has an associated Transaction Context field. For each packet transmitted, the DDM returns the packet's TransactionContext.

The LAN class is unusual in that there can be 0, 1, or many replies to a particular request. When the user receives a ***LanPacketSend*** or ***LanSduSend*** reply, it inspects each Transaction Context and matches the packet with a request. The user determines which sends are complete by using the TransactionContext data in the reply. The user must use the same InitiatorContext value for all ***LanPacketSend*** and ***LanSduSend*** requests.

6.10.2.2 Loopback of Transmitted Packets

Different medium types have different characteristics. One characteristic is the ability to receive the packet it sends, as specified in the table below.

Table 6-49. LAN Loopback Requirements

Media Type	Loopback Requirements
Ethernet	No Loopback
100BaseVG	No Loopback
Token Ring	Loopback
FDDI	Loopback
Fibre Channel	Loopback

Loopback means that when the DDM sends a packet containing a destination address normally received by the port, that packet is expected to appear in a receive bucket. The transmit control word of a send message may inhibit loopback for a particular send.

6.10.3 Receives

All received packets are transferred from the DDM using buckets. The initiator (e.g., the host LAN OSM) allocates memory, and describes this memory using a scatter-gather list. Each buffer marked in the scatter-gather list corresponds to a bucket. The DDM writes incoming packets into these buckets. Buckets do not have to be physically contiguous. The DDM describes each of the buckets it consumes, their order, and the location and length of each packet within those buckets by building a Packet Descriptor Block (PDB). The LAN device places the PDB in the reply message.

For media other than Fibre Channel, the LAN device places the packet in the bucket exactly as it arrived from the medium, including the MAC header information. For Fibre Channel, which generates MAC headers on transmission and transforms them on reception, bytes 0 through 7 of the buffer contain the destination MAC address, and bytes 8 through 15 contain the source MAC address. When IEEE 48-bit MAC addresses are used, they occupy the first six bytes of each field.

A packet can be returned in multiple buckets. This efficiently supports media with potentially large, but typically small packet sizes. If any bucket holding part of a packet is returned, all buckets containing any part of the packet must be returned. The user can require that a minimum number of bytes of a packet fit in the bucket (PacketOrphanLimit). For example, an OSM can specify that if a bucket contains the start of a packet, it contains at least the first *F* bytes.

The user specifies the amount of head space (PacketPrePad) the DDM leaves before the first packet in the first bucket and between subsequent packets. Each packet is preceded by the user-specified PacketPrePad field. This field is intended for OS use. The end of each packet is rounded up to a 32-bit boundary. The PacketPrePad is always in the same bucket as the start of the packet. If the PacketOrphanLimit prevents placing the next packet in a bucket, then the PacketPrePad is also placed in the next bucket.

A bucket must accommodate the PacketPrePad, plus at least PacketOrphanLimit bytes of the packet. In theory, a bucket could be as small as four bytes. In practice, a bucket should hold at least one typical packet for the medium in use.

The user tracks buckets using a BucketContext, analogous to the TransactionContext in messages, which is passed to the DDM in the scatter-gather list and reported back in the PDB. The Bucket Context is written into the scatter-gather list by the user.

When buckets are posted to the DDM, the DDM owns them. When a packet is received, the DDM (or its hardware) copies the packet into one or more buckets, depending on its size and the space remaining in a particular bucket. When the DDM reports a packet buffer back in the PDB, then ownership of the bucket returns to the user, and the DDM does not touch that bucket again unless it is reposted by the user.

The DDM can use buckets in arbitrary order.

The user can post buckets of varying sizes.

6.10.3.1 Posting Buckets

The ***LanReceivePost*** message posts a list of buckets, each marked by the end-of-buffer entry in the scatter-gather list (SGL). Each bucket must start on a 32-bit boundary and be an integral number of 32-bit words. The first SGL element (excluding ignore elements) of each bucket contains a BufferContext field. The first 32 bits of the BufferContext contains the BucketContext value for that bucket. The user recovers posted receive buckets using the ***UtilAbort*** message.

6.10.3.2 Indicating Receive Packets

A set of rules governs when a DDM indicates received packets (see section 6.10.7, *LAN Configuration and Operating Parameters* Batch Mode Control). Under low load, the DDM indicates a received packet immediately. Under a heavy load, the DDM collects received packets until a threshold is exceeded or a timer expires. In either case, the LAN device indicates the received packets with the ***LanReceivePost*** reply. This message lets the DDM provide a list of packets in the message payload.

6.10.3.2.1 Bucket Format

When a bucket has been (partially) filled it has the following format:

Table 6-50. Returned Bucket Format

PACKET BUFFER: Per-packet pattern, repeated for each packet in the bucket...		
Field Size	**Name**	**Description**
n x 32 bits	PacketPrePad	Size set by initiator using ***UtilParamsSet*** message. Always starts on a 32-bit boundary.
n x bytes	PacketData	Always starts on 32-bit boundary. Length is in the PDB, not here.
0-3 bytes	RoundUp	Round up to next 32-bit boundary – structure always starts and ends on a 32-bit boundary.
n x 32 bits	unused	Additional space before next Packet Buffer (i.e., next PacketPrePad). The length is arbitrary and determined by the DDM.

A bucket may begin with the end of a packet pattern from a previous bucket, and end with part of a packet pattern continued in the next bucket. In fact, the bucket may contain just an intermediate portion of a very large packet.

6.10.3.2.2 Packet Descriptor Block

A Packet Descriptor Block (*PDB*) is included in the reply message frame

The PDB contains a list of buckets and their packet descriptors. The packet descriptor indicates where this portion of the packet starts, the length of data, if the packet is continued from a previous bucket, if the packet is continued in the next bucket, if it is the last packet in the bucket, and if it is the last packet descriptor in the PDB.

Table 6-51. Packet Descriptor Block

Array of Bucket Descriptions		
32/64 bits	BucketContext (1st Bucket Descriptor)	This is the initiator (OSM) provided context field, which is simply copied out of the BufferContext field in the SGL. The size of this field is constant and either 32 or 64 bits.
32 bits	FirstPacketOffset	Low 24 bits are offset, in bytes, of the start of the packet data from the start of the bucket. PrePacketPad precedes this location if applicable. High 8 bits are flags, interpreted as: **`xxxx-xx00`** = valid packet **`xxxx-xx01`** = packet received with errors **`xxxx-x010`** = space to be skipped (the Packet Descriptor Block itself is in such a space) **`xxxx-000x`** = packet wholly contained in this bucket **`xxxx-x1xx`** = packet continued from previous bucket **`xxxx-1xxx`** = packet continued in next bucket **`x1xx-xxxx`** = last packet in this bucket **`11xx-0xxx`** = last packet in last bucket If a packet spans multiple buckets, multiple entries return, i.e., an entry for each bucket containing part of the packet.

Array of Bucket Descriptions		
32 bits	FirstPacketLength	Low 24 bits are Length, in bytes of the packet, excluding any PacketPrePad requested by the OS and not counting the RoundUp bytes. If a packet runs into next bucket, length is the number of bytes in this bucket. Next BucketDescriptor gives length of the rest of the packet. High eight bits are error status. The LAN device only indicates bad packets when requested (see parameter group 4, Media Operations) Error status values are: 00h = No Error 01h = Bad CRC 02h = Alignment Error 03h = Packet Too Long 04h = Packet Too Short 05h = Receive Overrun FFh = Other Error
32 bits	SecondPacketOffset	Same as FirstPacketOffset.
32 bits	SecondPacketLength	Same as FirstPacketLength.
(*n*-2) x 64 bits	3rd thru *n*th PacketOffset + PacketLength fields	3rd through *n*th PacketOffset and PacketLength fields – same as FirstPacketOffset and FirstPacketLength fields where *n*= number of packets in this bucket.
32/64 bits	BucketContext (2nd Bucket Descriptor)	As above
32 bits	FirstPacketOffset	As above
32 bits	FirstPacketLength	As above
32 bits	SecondPacketOffset	As above
32 bits	SecondPacketLength	As above
(*n*-2) x 64 bits	3rd thru *n*th PacketOffset + PacketLength fields	As above
3rd through *n*th Bucket Descriptor (as described above, i.e., BucketContext + number of PacketOffset & PacketLength fields) where *n*= number of Buckets		

6.10.3.3 Processing Receive Replies

The user must use the same InitiatorContext for all ***LanReceivePost*** requests. The user determines which buckets contain received packets and thus can be processed by the BucketContext in the reply's PDB. The PDB gives the user a list of packet buffers that contain the received packets in the order the DDM received them.

6.10.4 Batch and Error Control

Using the ***UtilParamsGet***and ***UtilParamsSet*** messages, the user can query and adjust various control parameters of the LAN port. Two sets of parameters that affect performance are *batch control* and *error control*.

Error control specifies which transaction errors to report in the transaction status. Since the protocol stacks above the user (and the user itself) use various timeouts on packets, it may be pointless to report most errors. Therefore, the DDM supports turning off reporting of individual transmission errors. If a packet encounters a transmission error when they are disabled, the transaction is reported as successful. Other errors, such as in the format of the packets or their batch list, are always reported.

Batch control specifies how to batch up received packets into buffers before replying to notify the user of their arrival. (Such notification does not imply an interrupt, since the IOP does message/interrupt batching in addition to the packet batching described here.) Under a light load, the DDM writes a few packets into each bucket and returns them quickly, to minimize latency. Under a heavy load, the DDM attempts to fill buckets with packets and report multiple, perhaps many, buckets with a single reply. Batch control specifies the load conditions when the DDM switches between batch and light load modes, and how much to batch in batch mode.

See Table 6-60 for more details.

6.10.5 Events

The LAN driver supports the generic events specified in section 6.1.3.4 for ***UtilEventRegister*** . In addition, the following LAN events are specified.

Table 6-52. LAN EventIndicator Assignments

Event Name	Bit	Description
LinkDown	0	The link to the physical medium is lost
LinkUp	1	The link to the physical medium is (re)established
MediaChange	2	The media changed (HDX/FDX, Line Rate, Connector type, etc.). The user should re-read the connector/connection type.

6.10.6 Messages

Table 6-53 shows requests that can be made of LAN class objects.

Table 6-53. Base LAN Class Request Messages

Function	Description
LanPacketSend	Send batch of packets
LanSduSend	Send batch of packets using Auto MAC Insertion
LanReceivePost	Post buckets to receive incoming packets

All LAN class messages are multiple-transaction messages. Typically, the MessageFlags field for requests should be set to 10h (for 32-bit context size) or 12h (for 64-bit context size). For a normal reply, it should contain D0h (for 32-bit context size) or D2h (for 64-bit context size). Since some requests provide a SGL, the value of the VersionOffset field depends on the SGL's location. Since some replies contain a TRL, the value of the VersionOffset field depends on the location of the TRL.

6.10.6.1 Packet Send

The ***LanPacketSend*** message sends an arbitrary number of packets, each of which might contain multiple memory fragments. The SGL is a scatter-gather list of fragments, in which the last fragment for any given packet is marked with the End-Of-Buffer bit. Each buffer is a packet to be transmitted. The first SGL element of each buffer contains a BufferContext field. The first 32 or 64 bits of the BufferContext contain the TransactionContext for the packet.

Unless the DDM indicated NoDaInSGL (in the TxModes in parameter group 7), the next 64 bits of BufferContext contain the Destination MAC address of the packet in media format (the order in which it is to be transmitted). Within the eight bytes of Destination Address when it is present, standard IEEE 48-bit MAC addresses occupy the first six bytes, followed by two bytes of zeros. The user and the DDM must support messages with DA in the SGL.

Each buffer contains a complete packet, exactly as it will transmit, including the MAC header.

<table>
<tr><th colspan="2">31 3 24</th><th colspan="2">23 2 16</th><th colspan="2">15 1 8</th><th colspan="2">7 0 0</th><th></th></tr>
<tr><td colspan="4">MessageSize</td><td colspan="2">MessageFlags</td><td colspan="2">VersionOffset</td><td>0</td></tr>
<tr><td colspan="2">***LanPacketSend***</td><td colspan="3">InitiatorAddress</td><td colspan="3">TargetAddress</td><td>4</td></tr>
<tr><td colspan="8">InitiatorContext</td><td>8</td></tr>
<tr><td colspan="8">TransmitControlWord</td><td>12 (16)</td></tr>
<tr><td colspan="8">SGL</td><td>16 (24)</td></tr>
</table>

offset values for 64-bit context field size in ()

Figure 6-71. *LanPacketSend* Request Message

Fields

TransmitControlWord — A set of flags that identify how this send request will execute. The Transmit Control Word is described in Table 6-54. All packets of a request transmit with the same attributes specified by TransmitControlWord

VersionOffset — A value of 41h for 32-bit context size and 61h for 64-bit context size.

Completion status can be reported directly by a reply to the send, or a single reply can complete multiple sends.

The memory described by the scatter-gather list can be reused by the user only after transmission completion is reported.

Table 6-54. Definition of the TransmitControlWord field

Bits	Description
Bits 2-0	**Access Priority**. Exact application is defined by particular medium type.
Bit 3	**Suppress CRC Generation**. This bit is set when the packet data contains the FCS field and the hardware does not generate that CRC value. This bit may be set only if CrcSuppression bit in the TxModes field of parameter group 7 is 1.
Bit 4	**Suppress Loopback**. This bit is set when receiving the transmitted packet is not desired. Useful during promiscuous receive mode and when transmitting broadcast messages on media that receive transmitted messages. This bit may be set only if LoopBackSuppression bit in the TxModes field of parameter group 7 is 1.
Bits 31-30	**Reply Mode:** Identifies the type of response requested by the host. Note that the latter two modes are valid only for messages that have immediate data.
0 0	Use batch rules for combining this reply with others
0 1	Reply as soon as transmission attempt is complete
1 0	Reply only if cannot transmit successfully
1 1	No reply required

Table 6-55. Packet Structures for Various Media

Media Type	Packet Structure
Ethernet/802.3	DA, SA, SDU, FCS
100BaseVG	DA, SA, SDU, FCS
Token Ring	AC, FC, DA, SA, SDU, FCS
FDDI	FC, DA, SA, SDU, FCS
Fibre Channel	SDU, FCS

DA Six-byte destination addresses field in format specified by AddressFormat in LAN Parameter Group 0000h

SA Six-byte source addresses field in format specified by AddressFormat in LAN Parameter Group 0000h

AC One-byte access control field

FC One-byte frame control field

SDU Variable-size MAC payload, includes the Routing Information field if present. For Ethernet/802.3 and 100BaseVG, the SDU includes the Type/Length field.

FCS Four-byte frame check sequence, only present if the *SUPPRESS CRC GENERATION* bit is set in the TransmitControlWord.

The reply structure for reporting transmission completion is shown in Figure 6-72.

3	2	1	0	
31 24	23 16	15 8	7 0	
MessageSize		MessageFlags	VersionOffset	0
Function	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TRL Control Word				12 (16)
ReqStatus	reserved	DetailedStatusCode		16 (24)
Transaction Context 1				20 (28)
Transaction Context 2				24 (36)
Transaction Context *n*				

Offset in () signifies offset for 64-bit context fields

Figure 6-72. *LanPacketSend* Reply Message

Fields

ReqStatus This field conveys the general status of the transaction, per Chapter 3.

DetailedStatusCode This field is reserved for a more detailed description of the status when required. Values for this field are defined in Table 6-56.

TransactionContext These are the TransactionContext values associated with each packet transmitted. All transactions reported in a single reply have the same ReqStatus and DetailedStatusCode. TransactionContext are not in any particular order.

VersionOffset A value of 51h for 32-bit context size and 71h for 64-bit context size.

If there is an error in transmission, and error reporting is suppressed, the packet is reported as successful and included with other successful transactions as appropriate.

If there is a transmission error in a packet and error reporting is not suppressed, then a separate reply is required to report the error. Error reports should be sent immediately and not batched.

Table 6-56. DetailedStatusCodes

Status	Description
LAN_SUCCESS	Success - no warnings
LAN_BAD_PACKET_DETECTED	Malformed packet detected
LAN_BUCKET_OVERRUN	DDM out of receive buckets.
LAN_CANCELED	The send or receive was canceled by the user.
LAN_DESTINATION_ADDRESS_DETECTED	A destination address was detected in the bucket context(s) of a Send Batch request message when it was not expected.
LAN_DESTINATION_ADDRESS_OMITTED	A destination address is required in the bucket context(s) of a Send Batch request message, but not present.
LAN_DESTINATION_NOT_FOUND	The destination specified in message not found
LAN_DEVICE_FAILURE	Device does not respond or responds with fault
LAN_DMA_ERROR	Error occurred when attempting to DMA information between Device and user
LAN_INVALID_TRANSACTION_CONTEXT	A buffer specified in the SGL did not contain a transaction context or the transaction context did not match a current transaction. May also indicate that the context field was the wrong size.
LAN_IOP_INTERNAL_ERROR	IOP detected an internal error
LAN_OUT_OF_MEMORY	IOP out of memory
LAN_PARTIAL_PACKET_RETURNED	The maximum number of buckets was returned in the Receive Reply message. The entire packet could not be described.
LAN_RECEIVE_ABORTED	Data receive aborted by user
LAN_RECEIVE_ERROR	Error on data receive
LAN_TRANSMISSION_ABORTED	Transmission of data aborted by user
LAN_TRANSMIT_ERROR	Error on transmit of data

6.10.6.2 SDU Send

The ***LanSduSend*** message sends an arbitrary number of packets, each of which might be composed of multiple memory fragments. The SGL is a scatter-gather list of fragments, in which the last fragment for any given packet is marked with the End-Of-Buffer bit. Each buffer is a packet to be transmitted. The first SGL element of each buffer contains a BufferContext field. The first 32 or 64 bits of the BufferContext contain the TransactionContext for the packet.

The next 64 bits of BufferContext contain the Destination MAC address of the packet in media format (the order in which it is to be transmitted). Within the eight bytes of Destination Address when it is present, standard IEEE 48-bit MAC addresses occupy the first six bytes, followed by two bytes of zeros.

Each buffer contains a packet, exactly as it is to be transmitted, excluding the MAC header. The MAC header is generated by the DDM. The Buffer Context always contains the destination MAC address of the packet in media format.

3 31 … 24	2 23 … 16	1 15 … 8	0 7 … 0	
MessageSize		MessageFlags	VersionOffset	0
LanSduSend	InitiatorAddress		TargetAddress	4
InitiatorContext				8
TransmitControlWord				12 (16)
SGL				16 (24)

offset values for 64-bit context field size in ()

Figure 6-73. ***LanSduSend*** **Request Message**

Fields

TransmitControlWord — A set of flags that identify how this send request will be executed. The Transmit Control Word is described in Table 6-54. All packets of a batch send transmit with the same attributes specified by TransmitControlWord.

VersionOffset — A value of 41h for 32-bit context size and 61h for 64-bit context size.

The behavior of the ***LanSduSend*** request is exactly the same as the ***LanPacketSend*** request. It uses same reply structure as shown in Figure 6-72 and the same DetailedStatusCode values in Table 6-56.

Table 6-57. Packet Structures for Various Media

Media Type	Packet Structure
Ethernet/802.3	SDU, FCS
100BaseVG	SDU, FCS
Token Ring	SDU, FCS
FDDI	SDU, FCS
Fibre Channel	SDU,FCS

SDU — Variable-size MAC payload, includes the Routing Information field if present. For Ethernet/802.3 and 100BaseVG, the SDU includes the Type/Length field.

FCS — Four-byte frame check sequence, only present if the *SUPPRESS CRC GENERATION* bit is set in the TransmitControlWord.

6.10.6.3 Post Receive Buckets

The scatter-gather list describes a set of buckets in memory. The end of each bucket is marked with an End-Of-Buffer entry, just as the ends of packets are for a send. The DDM fills buckets as described in section 6.10.3.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
LanReceivePost	InitiatorAddress	TargetAddress		4
InitiatorContext				8
BucketCount				12 (16)
SGL				16 (24)

offset values for 64-bit context field size in ()

Figure 6-74. *LanReceivePost* Request Message

VersionOffset A value of 41h for 32-bit context size and 61h for 64-bit context size.

6.10.6.4 Receive Reply

The reply message shown in Figure 6-75 is used to post buckets to the user. Posted buckets are described in 6.10.3.1.

31 3 24	23 2 16	15 1 8	7 0 0	
MessageSize		MessageFlags	VersionOffset	0
LanReceivePost	InitiatorAddress	TargetAddress		4
InitiatorContext				8
TrlFlags=80h	Reserved=00h	TrlElementSize=2	TrlCount	12 (16)
ReqStatus	reserved	DetailedStatusCode		16 (24)
BucketsRemaining				20 (28)
PdbArray (TRL)				24 (32)

offset values for 64-bit context field size in ()

Figure 6-75. *LanReceivePost* Reply Message

Fields

BucketsRemaining The running count of buckets that the DDM has left to consume. The host judges how badly the DDM needs more buckets by this field.

DetailedStatusCode If the DDM runs out of buckets, it posts an *OverRun* code.

PdbArray Array of Bucket Descriptors as described in Table 6-51.

VersionOffset A value of 61h for 32-bit context size and 81h for 64-bit context size.

6.10.7 LAN Configuration and Operating Parameters

Reading and modifying LAN parameters is performed by the ***UtilParamsGet*** and ***UtilParamsSet*** utility messages specified in 6.1.3.13. The list of parameter sets for the LAN class and their format are specified in the following tables.

Table 6-58. LAN Group 0000h - Device Information Parameter Group

GroupNumber	0000h
GroupType	SCALAR
Name	***LAN_DEVICE_INFO***
Description	Identifies the physical configuration of this LAN port.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	2 Bytes	LANType	Type of LAN. 0030h Ethernet 0040h 100base VG 0050h IEEE802.5/Token-Ring 0060h ANSI X3T9.5 FDDI 0070h Fibre Channel
1	r	2 Bytes	Flags	bit 0 0=physical LAN port 1=emulated LAN (ATM, FC) bit 1 0=simplex 1=full duplex
2	r	1 Byte	AddressFormat	The format of the address used. 00h = 48 Bit Universally administered IEEE address. Addresses are reported in media format. That is, the first byte transmitted is in the low order byte and the last two bytes are pad (0000h).
3	r	1 Byte	reserved1	reserved
4	r	2 Bytes	reserved2	reserved
5	r	4 Bytes	MinPacketSize	Minimum size of a packet, as seen by the initiator.
6	r	4 Bytes	MaxPacketSize	Maximum packet size (number of bytes including MAC header). The DDM treats a received packet greater than this size as follows: If passing error packets up, then truncates packet at MaxPacketSize bytes and sets error = *PACKET_TOO_LONG*. Otherwise discards packet. Maximum size including MAC header.
7	r	8 Bytes	HardwareAddress	MAC address stored permanently in the adapter.
8	r	8 Bytes	MaxTxWireSpeed	Maximum transmission speed (in bps) of the port.
9	r	8 Bytes	MaxRxWireSpeed	Maximum receive speed (in bps) of the port.

Table 6-59. LAN Group 0001h - MAC Address Parameter Group

GroupNumber	0001h
GroupType	SCALAR
Name	***LAN_MAC_ADDRESS***
Description	Provides MAC Address administration and Reception attributes.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r/w	8 bytes	ActiveAddress	This is the MAC address in media format that the card is listening for. The last two bytes are padding.
1	r/w	8 Bytes	CurrNetworkAddress	MAC address that can be modified by software, referred to as the Locally Administered Address (LAA).
2	r/w	8 Bytes	FunctionalAddressMask	A bit-specific address, in MAC address format, specifying the functional address mask for media that support functional addressing such as Token Ring.
3	r/w	4 bytes	FilterMask	Broadcast, Multicast, Functional Addresses, packet types, MAC packets, UserData packets, etc. May be read only. See Table 6-83.
4	r	4 bytes	HardwareFilterMask	Mask indicating which of the FilterMask capabilities are implemented in hardware.
5	r	4 bytes	MaxSizeMulticastTable	Number of multicast addresses.
6	r	4 bytes	MaxFilterPerfect	Maximum number of multicast addresses for which perfect filtering can be done
7	r	4 bytes	MaxFilterImperfect	Maximum number of multicast addresses for which imperfect filtering can be done

Table 6-60. LAN Parameter Group 0002h - Multicast MAC Address Table

GroupNumber	0002h
GroupType	TABLE
Name	***LAN_MULTICAST_ MAC_ADDRESS***
Description	This table specifies all the Multicast Addresses that the LAN Adapter is filtering on.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r/w	**8 bytes**	MulticastMACAddress	Multicast MAC address in media format. The last two bytes are padding.

Table 6-61. LAN Parameter Group 0003h - Batch Control

GroupNumber	0003h
GroupType	SCALAR
Name	***LAN_BATCH_CONTROL***
Description	Controls whether and when the DDM switches from batch mode to immediate mode and back. In immediate mode, a reply reports to the host every packet's arrival as soon as the packet is complete. In batch mode, packets are batched up and notification is sent for the group.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r/w	4 bytes	BatchFlags	Bit-specific field: Bit 0 = If true receive batch mode is disabled, always return one packet per bucket and post reply immediately. A 0 indicates receive batch mode is enabled. Bit 1 = True if current receive mode is batch mode, else false. Bit 2 = True if receive batch mode has been forced on, 0 indicates automatic switching in and out of batch mode when enabled.
1	r/w	4 bytes	RisingLoadDelay	When the received packet rate exceed RisingLoadThreshold for longer than RisingLoadDelay (in 10ms), turn on batch mode.
2	r/w	4 bytes	RisingLoadThreshold	See RisingLoadDelay
3	r/w	4 bytes	FallingLoadDelay	When the received packet rate falls below the FallingLoadThreshold for longer than FallingLoadDelay (in 10ms), switch out of batch mode into immediate post mode.
4	r/w	4 bytes	FallingLoadThreshold	See FallingLoadDelay
5	r/w	4 bytes	MaxBatchCount	When in batch mode, post notification when this many packets have been received. Ignore if 0.
6	r/w	4 bytes	MaxBatchDelay	When in batch mode, post notification any time a received packet has been held for this long. In milliseconds. Ignore if 0. A delay counter is reset to this value whenever packet notification is posted. It starts counting down whenever a packet is received. If it ever goes off, all packets in hand are posted.
7	r/w	4 bytes	TransCompDelay	Transmission Completion Reporting delay, in milliseconds. Similar behavior to MaxBatchDelay, but it decides whether to piggyback transmission completion onto a receive or post it directly.

Table 6-62. LAN Parameter Group 0004h

GroupNumber	0004h
GroupType	SCALAR
Name	***LAN_OPERATION***
Description	Manages how the DDM responds to the user

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r/w	4 bytes	PacketPrePad	Length of per-packet prepad, number of 32-bit words.
1	r/w	4 bytes	UserFlags	Bit 0: (T/F) If true then transmission error reporting is turned on, and any transmission error is reported. If false then transmission error reporting is off, and such errors are ignored and the packet reported as if no error occurred. Only errors involving ill-formed packets and the like are reported.
2	r/w	4 bytes	PacketOrphanLimit	If an entire packet does not fit in a bucket, then at least PacketOrphanLimit bytes of the packet must appear in the first bucket that any part of the packet is in. This value includes the PacketPrePad. 0 is a legal value.

Table 6-63. LAN Parameter Group 0005h

GroupNumber	0005h
GroupType	SCALAR
Name	***LAN_MEDIA_OPERATION***
Description	Manipulates media specific items. Different structures are defined for various media types (Ethernet, token ring, etc.) which manage how the DDM responds to the user.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	32	ConnectorType	Type of connector being used to physically attach this port to the LAN. 0 OTHER 1 UNKNOWN 2 AUI 3 UTP 4 BNC 5 RJ45 6 STP DB9 7 Fiber MIC 8 Apple AUI 9 MII 10 Copper DB9, 1.0625 GBd 11 Copper AW{ HSSDC, 1.0625 Gdb 12 Optical LW, 1.0625 Gbd 100-SM-LL_L 13 Serial Interface Protocol 14 Optical SW, 1.0625 Gbd 100-M5-SN-I
1	r	4 bytes	ConnectionType	See Table 6-64
2	r	8 bytes	CurrentTxWireSpeed	Actual transmission speed, in bits per second, of the physical connection. LAN emulation over other media should report the actual speed here, not the speed of the emulated medium.
3	r	8 bytes	CurrentRxWireSpeed	Actual receive speed in bits per second of the physical connection. LAN emulation over other media should report the actual media speed here, not the speed of the emulated medium.
4	r	1 byte	FullDuplexMode	0 = HDX; -1=FDX
5	r	1 Byte	LinkStatus	Status of the medium. 0 UNKNOWN (initializing, true state not yet known) 1 Normal 2 Failure 3 Reset, Recovered 255 Other
6	r/w	1 byte	BadPacketHandling	Flags whether the host wishes to see bad packets. If set to -1, the DDM will pass all bad packets, including those with CRC errors, alignment errors, and runt and oversized packets up to the host.

Table 6-64. LAN Connection Types

0000h	UNKNOWN
Ethernet Types:	
0301h	AUI
302h	10Base5
0303h	FOIRL
0304h	10BASE2
0305h	10BROAD36
0306h	10BASE-T
0307h	10BASE-FP
0308h	10BASE-FB
0309h	10BASE-FL
030Ah	100BASE-TX
030Bh	100BASE-FX
030Ch	100BASE-T4
100BaseVG Types	
0401h	100BASE-VG
Token Ring Types	
0501h	4 Mbit/Sec
0502h	16 Mbit/Sec
FDDI Types	
0601h	125 Mbaud/Sec fiber
Fibre Channel Types	
0701h	Point to Point
0702h	Arbitrated Loop
0703h	Public Loop
0704h	Fabric
Other Types	
0F00h	Emulation over other media
0F01H	Other

Table 6-65. LAN Parameter Group 0006h

GroupNumber	0006h
GroupType	TABLE - optional
Name	***LAN_ALTERNATE_ ADDRESS***
Description	This table specifies all the unicast MAC addresses that the LAN Adapter is filtering on.

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	AlternateAddress	Alternate address.

Table 6-66. LAN Parameter Group 0007h

GroupNumber	0007h
GroupType	SCALAR
Name	***LAN_TRANSMIT_INFO***
Description	Identifies the Port's Transmit attributes

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 bytes	TxMaxPacketSG	The maximum number of Scatter Gather entries allowed per packet. This is a hard limit.
1	r	4 bytes	TXMaxChainSG	The maximum number of Scatter Gather entries allowed in a chained block. This is a hard limit..
2	r	4 bytes	TXMaxPktsOut	The maximum number of outstanding packets. This is a soft limit.
3	r	4 bytes	TXMaxReqPkts	The maximum number of packets per transmit request. This is a soft limit.
4	r	4 bytes	TxModes	This is a bit specific field: Bit 0 reserved. Bit 1: NoDaInSGL - when set, this device does not requre the DA to be in the Buffer Context field for ***LanPacketSend*** requests. Bit 2: CrcSuppression - When this bit is set, this device supports suppression of the CRC. Otherwise CRC is always generated by hardware. Bit 3: LoopSuppression - When this bit is set, this device supports suppression of receiving the transmitted packet. Otherwise the packet loop backs as defined by the media type. Bit 4: MAC Insertion - when set, this device can accept ***LanSduSend*** requests and generate the MAC header. Bit 5: RIF Insertion - when set, this device can accept ***LanSduSend*** requests and generate the MAC header and the Routing Information Field. Bit 6: IPChecksum - when set, this device is capable of calculating IP checksum. Other bits reserved

Table 6-67. LAN Parameter Group 0008h

GroupNumber	0008h
GroupType	SCALAR
Name	***LAN_RECEIVE_ INFO***
Description	Identifies the Port’s Receive attributes

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	4 bytes	RXMaxChain	The maximum size of a chain element in a receive bucket. This is a hard limit.
1	r	4 bytes	RXMaxBuckets	The maximum number of receive buckets. This is a soft limit.

Table 6-68. LAN Parameter Group 0100h

GroupNumber	0100h
GroupType	SCALAR - Required
Name	***LAN_HISTORICAL_STATS***
Description	**LAN Historical Statistics**

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	TotalPacketsTransmitted	Total packets transmitted.
1	r	8 bytes	TotalBytesTransmitted	Total bytes transmitted. Actual number of bytes between starting and ending delimiters.
2	r	8 bytes	TotalPacketsReceived	Total packets received.
3	r	8 bytes	TotalBytesReceived	Total bytes received. Actual number of bytes between starting and ending delimiters.
4	r	8 bytes	TotalTransmitErrors	Total transmit errors of any type.
5	r	8 bytes	TotalReceiveErrors	Total receive errors of any type.
6	r	8 bytes	ReceiveNoBuffer	Number of packets dropped due to lack of receive buffers
9	r	8 bytes	AdapterResetCount	Number of times the adapter was reset due to internal failure or external request.

Table 6-69. LAN Parameter Group 0180h

GroupNumber	0180h
GroupType	SCALAR
Name	**LAN_SUPPORTED_OPTIONAL_HISTORICAL_STATS**
Description	Specifies which statistics are supported

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	StatisticsSupported	Each bit (0-63) corresponds to the field index of the Optional Historical Statistics group 0181h. 0 Statistic not supported 1 Statistic supported

Table 6-70. LAN Parameter Group 0181h

GroupNumber	0181h
GroupType	SCALAR
Name	**LAN_OPTIONAL_HISTORICAL_STATS**
Description	

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	TxRetryCount	Number of times a transmit was retried due to some sort of failure
1	r	8 bytes	ReceiveCRCErrorCount	Number of packets received with a CRC error
2	r	8 bytes	DirectedBytesTx	Number of bytes transmitted to a specific MAC address
3	r	8 bytes	DirectedPacketsTx	Number of packets transmitted to a specific MAC address
4	r	8 bytes	MulticastBytesTx	Number of bytes transmitted to a multicast address
5	r	8 bytes	MulticastPacketsTx	Number of packets transmitted to a multicast address
6	r	8 bytes	BroadcastBytesTx	Number of bytes transmitted to a broadcast address
7	r	8 bytes	BroadcastPacketsTx	Number of packets transmitted to a broadcast address
8	r	8 bytes	DirectedBytesRx	Number of bytes received with a specific MAC destination address
9	r	8 bytes	DirectedPacketsRx	Number of packets received with a specific MAC destination address
10	r	8 bytes	MulticastBytesRx	Number of bytes received with a multicast destination address
11	r	8 bytes	MulticastPacketsRx	Number of packets received with a multicast destination address
12	r	8 bytes	BroadcastBytesRx	Number of bytes received with a broadcast destination address
13	r	8 bytes	BroadcastPacketsRx	Number of packets received with a broadcast destination address
7	r	8 bytes	TotalGroupAddrTxCount	Total packets transmitted to a group destination address.
8	r	8 bytes	TotalGroupAddrRxCount	Total packets received at a group destination address.

Table 6-71. LAN Parameter Group 0200h

GroupNumber	0200h
GroupType	SCALAR
Name	**LAN_802_3_HISTORICAL_STATS**
Description	

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	RxAlignmentError	Number of misaligned packets received
1	r	8 bytes	TxOneCollision	Number of packets transmitted that experienced a single collision
2	r	8 bytes	TxMultipleCollisions	Number of packets transmitted that experienced multiple collisions
3	r	8 bytes	TxDeferred	Number of packets transmitted OK after deferral
4	r	8 bytes	TxLateCollision	Number of packets transmitted that experienced a collision outside the nominal window
5	r	8 bytes	TxMaxCollisions	Number of packets transmitted that had to be restarted due to maximum collisions
6	r	8 bytes	TxCarrierLost	Number of packets transmitted in which carrier sense was not present or was lost during transmission
7	r	8 bytes	TxExcessiveDeferrals	Total packets transmitted that experience excessive deferrals

Table 6-72. LAN Parameter Group 0280h

GroupNumber	0280h
GroupType	SCALAR
Name	**LAN_SUPPORTED_802_3_HISTORICAL_STATS**
Description	Specifies which statistics are supported

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	StatisticsSupported	Each bit (0-63) corresponds to the FieldIdx of the Optional Historical Statistics group 0281h. 0 Statistic not supported 1 Statistic supported

Table 6-73. LAN Parameter Group 0281h

GroupNumber	0281h
GroupType	SCALAR
Name	**LAN_OPTIONAL_802_3_HISTORICAL_STATS**
Description	

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	RxOverrun	Number of times the adapter could not keep up with the wire on receive (usually because the IOP was too slow reading data), causing a dropped packet.
1	r	8 bytes	TxUnderrun	Number of times the adapter could not keep up with the wire on transmit (usually because the IOP was too slow writing data), causing transmission of a packet with bad CRC
2	r	8 bytes	TxHeartbeatFailure	Number of packets transmitted without a valid heartbeat (linkbeat)

Table 6-74. LAN Parameter Group 0300h

GroupNumber	0300h
GroupType	SCALAR
Name	**LAN_802_5_HISTORICAL_STATS**
Description	

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	LineErrors	Number of packets with invalid Frame Check Sequence or with a code violation
1	r	8 bytes	LostFrames	Number of packets transmitted that failed to return
2	r	8 bytes	ACError	Number of SMP packets received with invalid A and C values
3	r	8 bytes	TxAbortDelimiter	Number of times an abort delimiter is transmitted
4	r	8 bytes	BurstErrors	Number of burst-five errors detected
5	r	8 bytes	FrameCopiedErrors	Number of directed packets received with FS field bits set to 1
6	r	8 bytes	FrequencyErrors	Number of times the incoming signal frequency is out of range
7	r	8 bytes	InternalError	Number of recoverable internal errors
8	r	8 bytes	LastRingStatus	Last reported ring status, with the following bit values: Bit15: Signal Loss Bit14: Hard Error Bit13: Soft Error Bit12: Transmit Beacon Bit11: Lobe Wire Fault Bit10: Auto-Removal Error 1 Bit9: Reserved Bit8: Remove Received Bit7: Counter Overflow Bit6: Single Station Bit5: Ring Recovery Bit4-0: Reserved
9	r	8 bytes	TokenError	Number of times the active monitor detects TVX timer expiration
10	r	8 bytes	UpstreamNodeAddress	MAC Address of the upstream node (in the low order 6 bytes of the field, in line order, low order byte first)
11	r	8 bytes	LastRingID	Value of the local ring
12	r	8 bytes	LastBeaconType	Value of the last beacon type

Table 6-75. LAN Parameter Group 0380h

GroupNumber	0380h
GroupType	SCALAR
Name	**LAN_SUPPORTED_802_5_HISTORICAL_STATS**
Description	Specifies which statistics are supported

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	StatisticsSupported	Each bit (0-63) corresponds to the FieldIdx of the Optional Historical Statistics group 0381h. 0 Statistic not supported 1 Statistic supported

Table 6-76. LAN Parameter Group 0381h

GroupNumber	0381h
GroupType	SCALAR
Name	**LAN_OPTIONAL_802_5_HISTORICAL_STATS**
Description	Place holder only. No optional statistics defined for 802.5

Table 6-77. LAN Parameter Group 0400h

GroupNumber	0400h
GroupType	SCALAR
Name	**LAN_FDDI_HISTORICAL_STATS**
Description	

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	ConfigurationState	ANSI fddiSMTCFState, with the following values: 0: Isolated 1: Local_a 2: Local_b 3: Local_ab 4: Local_s 5: wrap_a 6: wrap_b 7: wrap_ab 8: wrap_s 9: c_wrap_a 10: c_wrap_b 11: c_wrap_s 12: through
1	r	8 bytes	UpstreamNode	ANSI fddiMACUpstreamNbr (0 if unknown)
2	r	8 bytes	DownstreamNode	ANSI fddiMACDownstreamNbr (0 if unknown)
3	r	8 bytes	FrameErrors	Number of packet errors detected by this MAC which were not already detected by another MAC
4	r	8 bytes	FramesLost	Number of times a packet was stripped on reception due to format errors
5	r	8 bytes	RingMgmtState	Current value of the Ring Management State, with the following values: 0: Isolated 1: Non-Op 2: Rind-Op 3: Detect 4: Non-Op-Dup 5: Ring-Op-Dup 6: Directed 7: Trace
	r	8 bytes	LCTFailures	Number of times the Link Confidence Test failed during connection management
7	r	8 bytes	LEMRejects	Number of times a link is rejected
8	r	8 bytes	LEMCount	Aggregate Link Error Monitor error count
9	r	8 bytes	LConnectionState	Current State of this port's PCM state machine, with the following values: 0: Off 1: Break 2: Trace 3: Connect 4: Next 5: Signal 6: Join 7: Verify 8: Active 9: Maintenance

Table 6-78. LAN Parameter Group 0480h

GroupNumber	0480h
GroupType	SCALAR
Name	**LAN_SUPPORTED_FDDI_HISTORICAL_STATS**
Description	Specifies which statistics are supported

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	StatisticsSupported	Each bit (0-63) corresponds to the FieldIdx of the Optional Historical Statistics group 0481h. 0 Statistic not supported 1 Statistic supported

Table 6-79. LAN Parameter Group 0481h

GroupNumber	0481h
GroupType	SCALAR
Name	**LAN_OPTIONAL_FDDI_HISTORICAL_STATS**
Description	Place holder only. No optional statistics defined for FDDI

Table 6-80. LAN Parameter Group 0500h

GroupNumber	0500h
GroupType	SCALAR
Name	**LAN_FC_HISTORICAL_STATS**
Description	Place holder only. No required statistics defined for Fibre Channel

Table 6-81. LAN Parameter Group 0580h

GroupNumber	0580h
GroupType	SCALAR
Name	**LAN_SUPPORTED_FC_HISTORICAL_STATS**
Description	Specifies which statistics are supported

FieldIdx	(r/w)	Field Size	Parameter Name	Description
0	r	8 bytes	StatisticsSupported	Each bit (0-63) corresponds to the FieldIdx of the Optional Historical Statistics group 0581h. 0 Statistic not supported 1 Statistic supported

Table 6-82. LAN Parameter Group 0581h

GroupNumber	0581h
GroupType	SCALAR
Name	**LAN_OPTIONAL_FC_HISTORICAL_STATS**
Description	Place holder only. No optional statistics defined for Fibre Channel

Table 6-83. FilterMask

Bits	Parameter Name	Description
0	UserDataFramesDisable	0=reception of user data packets is enabled. 1=disabled
1	PromiscuousEnable	0=normal 1=receive all packets, regardless of their address
2	PromiscuousMcEnable	0=normal 1=receive all multicast group packets regardless of their address
8	BroadcastDisable	0=reception of broadcast packets is enabled 1=disabled
9	MulticastDisable	0=reception of multicast group packets is enabled 1=disabled
10	FunctionalAddressesDisable	0=reception of functional addresses is enabled 1=disabled
12,11	MACreportingMode	0,0 = Do not pass MAC packets to user
		0,1 = Pass only priority MAC packets to user
		1,0 = Pass all received MAC packets to user
		1,1 = Receive all MAC packets regardless of their address and pass to the user

6.11 WAN Class

Not defined at this time – to be supplied at a later date.

6.12 Fibre Channel Class

Not defined at this time – to be supplied at a later date.

A.
Differences from Previous Version

This appendix identifies the main differences from version 1.0 of this specification.

A.1 Technical Changes

For consistency, the term *adapter* is used for controllers and add-in cards. The term *I_2O object* was changed to *I_2O device* or *device*.

Added detail to DDM upgrading safeguards.

Added missing details to configuration Dialogue.

Changed usage of InitiatorAddress and TargetAddress fields in reply message such that the target no longer swaps addresses for replies.

Modified the message for installing and loading DDMs to include downloading/storage of any software module including the IRTOS itself. Also provided for uploading software modules.

Enhanced sections on IOP and DDM initialization. Provided for BIOS initialization and OS takeover. Enhanced IOP status to include information needed prior to full OS I_2O drivers being loaded.

Provided for both 32-bit and 64-bit context values.

Enhanced requirements for PCI operation that eliminate the potential offset between an IOP's PCI bus and the system bus. Also redefined local memory access attributes so DDMs can allocate appropriate memory based on its ability to be accessed by adapters.

Changed from the host identifying public address space such that private space was all else to the host expressly specifying private space.

Defined additional optional IOP facilities/capabilities such as battery-backed RAM for data caches and non-volatile memory.

Extensive changes to the scatter-gather list. Changes include: use of local address, immediate data and bit bucket elements, non data elements, transport element, SGL attributes (e.g., foreign page frame sizes, hints), and adding direction bit for data buffers.

Defined a formal Transaction Reply List for replying to multiple transactions with a single reply using multiple formats. Modified LAN class receive PDB (moved bit positions for LastElement and LastTransaction).

Changed Transport Status in message header to indicate SGL/TRL location in message.

Changed default request and reply structures from recommended to required and moved them from chapter 6 to chapter 3. Transaction context required in fixed location(s).

Added message status to indicate dirty abort and faults as well as process aborts.

Enhanced Claim process to support a single primary user with optional service for alternate users, peer service, and management service.

Enhanced Params Get/Set operations to include tables and allow operation on individual fields. Replaced messages such as Identify and GetFamily utility messages with definition of parameter groups.

Reduced requirement of number of outstanding message frames for inbound queue.

Added PCI system interrupt capability to IOP.

Added executive class message to read configuration registers of hidden devices. Also added configuration validation message. Revised load and install messages adding an upload message. Changed SysTabSet and SysModify commands to indicate private space instead of public space.

Made the host's page frame size the default size for all IOPs.

Specified requirement for IOP to maintain certain information across resets.

Added fields to DDM's module header.

Defined TCL scripts for configuration dialog, and defined TCL Script table in module's header.

Defined private message class mechanism.

The following utility messages have been removed: ConfigResponse, GetExtensions, GetFamilyID, Identify, GetParamsImmediate, SetParamsImmediate.

Expanded UtilAbort to allow wildcard mataches and specify only clean aborts. Redefined UtilClaim message. Added page number to UtilConfigDialog message.

Tape class defined in Chapter 6.

All Message classes enhanced.

Event Indicator assignments for the ***UtilEventRegister*** (old ***EventNotify*)** message have been reapportioned such that only the first 16 bits are available for class-specific assignment. This affects Block storage and Tape class assignment. Additional generic assignment have been added.

A.2 Naming Conventions

All field names with an underscore (_) had the underscore removed. Constants are listed in all UPPERCASE and use underscore for readability.

Applying the noun/verb naming convention changed the name of most messages. The following tables correlate the old message names with the new names.

A.2.1 Message Names

Class	Old Name	New Name
Executive		***ExecAdapterRead (new)***
Executive		***ExecConfigValidate (new)***
Executive		***ExecSwUpload (new)***
Executive	***AssignDevice***	***ExecAdapterAssign***
Executive	***AssignObject***	***ExecDeviceAssign***
Executive	***ClearIOP***	***ExecIopClear***
Executive	***ConnectIOP***	***ExecIopConnect***
Executive	***ConnSetup***	***ExecConnSetup***
Executive	***CreateStaticMF***	***ExecStaticMfCreate***
Executive	***DestroyDDM***	***ExecDdmDestroy***
Executive	***EnableDDM***	***ExecDdmEnable***
Executive	***EnablePath***	***ExecPathEnable***
Executive	***EnableSys***	***ExecSysEnable***
Executive	***EventNotify***	***UtilEventNotify***
Executive	***GetHRT***	***ExecHrtGet***
Executive	***GetStatus***	***ExecStatusGet***
Executive	***GetXCT***	***ExecXctGet***
Executive	***InitOutbound***	***ExecOutboundInit***
Executive	***InstallDDM***	***ExecSwDownload (revised)***
Executive	***LCTNotify***	***ExecLctNotify***
Executive	***LoadDDM***	***ExecSwDownload (revised)***
Executive	***ModifySys***	***ExecSysModify***
Executive	***QuiesceDDM***	***ExecDdmQuiesce***
Executive	***QuiescePath***	***ExecPathQuiesce***
Executive	***QuiesceSys***	***ExecSysQuiesce***
Executive	***ReleaseDevice***	***ExecAdapterRelease***
Executive	***ReleaseObject***	***ExecDeviceRelease***
Executive	***ReleaseStaticMF***	***ExecStaticMfRelease***
Executive	***RemoveDDM***	***ExecSwRemove (revised)***
Executive	***ResetDDM***	***ExecDdmReset***
Executive	***ResetIOP***	***ExecIopReset***
Executive	***ResetPath***	***ExecPathReset***
Executive	***SetBiosInfo***	***ExecBiosInfoSet***
Executive	***SetBootDevice***	***ExecBootDeviceSet***
Executive	***SetSysTab***	***ExecSysTabSet***
Executive	***SuspendDDM***	***ExecDdmSuspend***

Class	Old Name	New Name
Utility	***NOP***	***UtilNOP***
Utility	***Abort***	***UtilAbort***
Utility	***Identify***	see UtilParamsGet
Utility	***GetFamilyID***	see UtilParamsGet
Utility	***GetExtensions***	see UtilParamsGet
Utility	***GetParams***	***UtilParamsGet***
Utility	***GetParamsImm***	see UtilParamsGet
Utility	***SetParams***	***UtilParamsSet***
Utility	***SetParamsImm***	see UtilParamsSet
Utility	***MessageFail***	***UtilReplyFaultNotify***
Utility	***Claim***	***UtilClaim***
Utility	***ClaimRelease***	***UtilClaimRelease***
Utility	***Reserve***	***UtilReserve***
Utility	***ReserveRelease***	***UtilReserveRelease***
Utility	***EventNotify***	***UtilEventRegister***
Utility	***EventAck***	***UtilEventAck***
Utility	***ConfigDialog***	***UtilConfigDialog***
Utility	***ConfigResponse***	not needed

Class	Old Name	New Name
DDM	***ConnModify***	See DdmDeviceReset
DDM	***ConnResume***	See DdmDeviceResume
DDM	***ConnSuspend***	See DdmDeviceSuspend
DDM	***DDM_Reset***	***DdmSelfReset***
DDM	***DDM_Resume***	***DdmSelfResume***
DDM	***DDM_Suspend***	***DdmSelfSuspend***
DDM	***DeviceAttach***	***DdmAdapterAttach***
DDM	***DeviceReconfig***	***DdmAdapterReconfig***
DDM	***DeviceRelease***	***DdmAdapterRelease***
DDM	***DeviceResume***	***DdmAdapterResume***
DDM	***DeviceSuspend***	***DdmAdapterSuspend***
DDM	***ObjectAttach***	***DdmDeviceAttach***
DDM	***ObjectRelease***	***DdmDeviceRelease***
DDM	***Path_Reset***	***DdmDeviceReset***
DDM	***Path_Resume***	***DdmDeviceResume***
DDM	***Path_Suspend***	***DdmDeviceSuspend***
DDM	***SystemChange***	***DdmSystemChange***
DDM	***SystemEnable***	***DdmSystemEnable***
DDM	***SystemHalt***	***DdmSystemHalt***

A.2.2 Field Name Changes

Class	Old Name	New Name
	SG_List	SGL
	MsgFlags	MessageFlags
	Version/Status	VersionOffset

Index