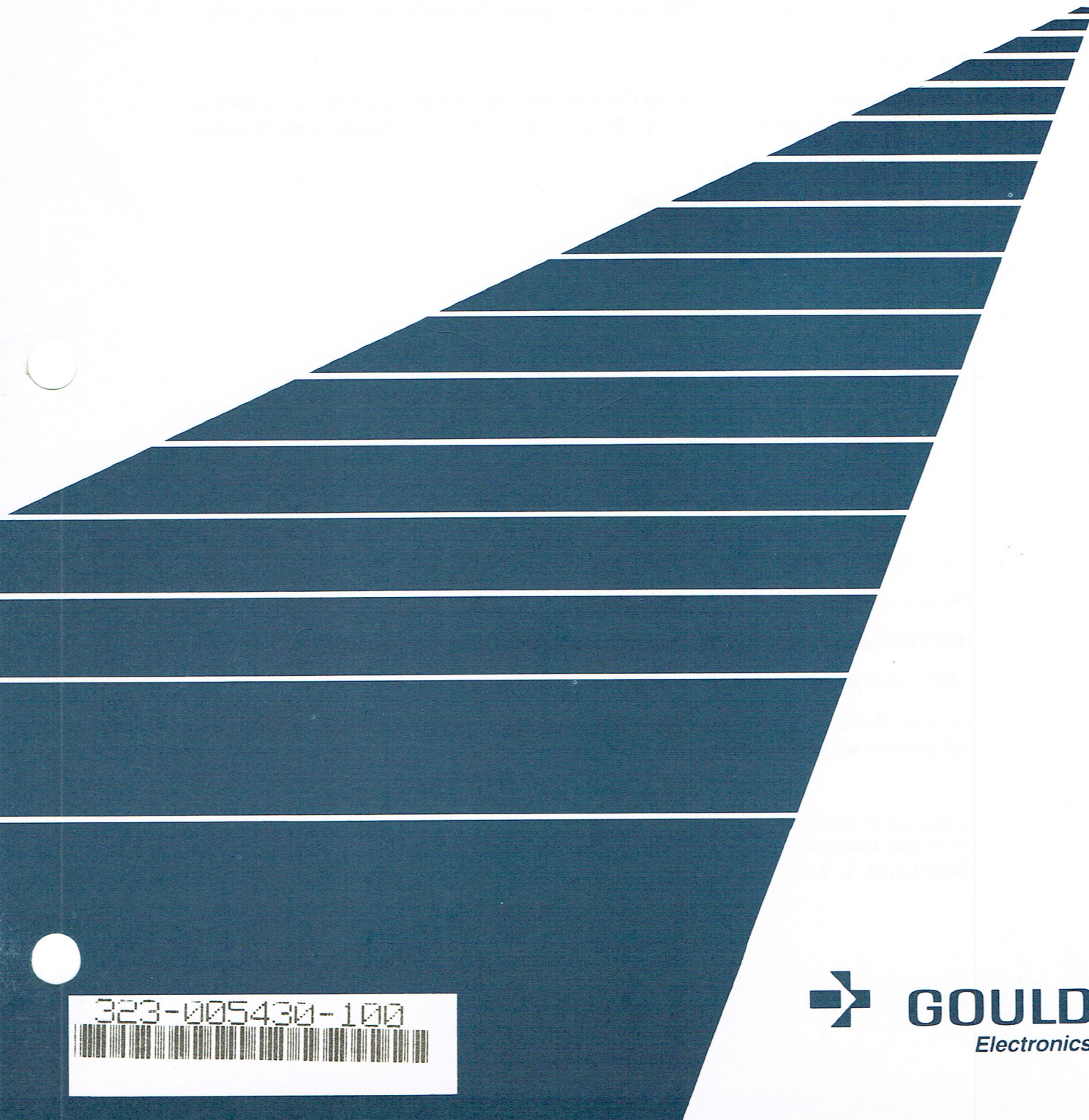



CPL

UTX/32™ Release 2.1

Input/Output Subsystem Guide

January 1988



323-005430-100




GOULD
Electronics

Limited Rights

This manual is supplied without representation or warranty of any kind. Gould Inc. therefore assumes no responsibility and shall have no liability of any kind arising from the supply or use of this publication or any material contained herein.

Proprietary Information

The information contained herein is proprietary to Gould CSD and/or its vendors, and its use, disclosure or duplication is subject to the restrictions stated in the Gould CSD license agreement Form No. 620-06 or the appropriate third-party sublicense agreement.

Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b) (3) (ii) of the rights in Technical Data and Computer Software Clause at 52.277.7013.

Gould Inc., Computer Systems Division
6901 West Sunrise Boulevard
Fort Lauderdale, Florida 33313

PowerNode, SelBUS, and UTX/32 are trademarks of Gould Inc.

Ethernet is a trademark of Xerox Corporation.

HYPERchannel is a registered trademark of Network Systems Corporation.

UNIX is a registered trademark of AT&T Bell Laboratories.

Portions of the UTX/32 Operating System are proprietary to AT&T Bell Laboratories, and portions are proprietary to Gould CSD.

Copyright © 1988 by Gould Inc.
All Rights Reserved
Printed in the U.S.A.

History

The *Input/Output Subsystem Guide for Gould UTX/32*, Release 2.0, Publication Order Number 323-005430-000, was printed in September, 1986.

The *UTX/32 Input/Output Subsystem Guide*, Release 2.1, Publication Order Number 323-005430-100, was printed in January, 1988.

This document contains the following pages:

Title page
Copyright page
History page, page iii/iv
Contents, pages v through vii
Figures, page viii
Chapter 1, pages 1-1 through 1-3/1-4
Chapter 2, pages 2-1 through 2-21/2-22
Chapter 3, pages 3-1 through 3-8
Chapter 4, pages 4-1 through 4-11/4-12
Chapter 5, pages 5-1 through 5-4
Appendix A, pages A-1 through A-25/A-26
References, page RF-1/RF-2

Contents

	Figures	viii
1	Introduction	1-1
1.1	Scope and Purpose of this Guide	1-1
1.2	Summary of Contents	1-1
1.3	Reader Prerequisites	1-2
1.4	Typographic Conventions	1-2
2	The Input/Output Interface	2-1
2.1	Introduction	2-1
2.2	Interfaces	2-2
2.2.1	Device Driver Interface	2-2
2.2.2	Device Interface	2-3
2.2.3	Maintenance Interface	2-3
2.2.4	System Configuration Interface	2-5
2.3	System Initialization	2-5
2.3.1	Logging	2-5
2.4	Data Structures	2-7
2.4.1	Processor Scratchpad (SCR) Entries	2-7
2.4.2	Service Interrupt Vector (SIV)	2-9
2.4.3	Interrupt Context Block (ICB)	2-9
2.4.4	Device Control Block (DCB)	2-9
2.4.5	Device Parameter Table (DPT)	2-10
2.4.6	Subchannel Interrupt Routing Table (SRT)	2-11
2.4.7	Minor-device-to-DCB Mapping Table (MTD)	2-11
2.5	IOI Execution Paths	2-11
2.5.1	Calls from the Device Driver	2-11
2.5.2	Device Interrupts	2-12
2.6	IOI Entry Points	2-12
2.6.1	<code>ioi_init</code>	2-13
2.6.2	<code>ioi_phys</code>	2-13
2.6.3	<code>ioi_ictl</code>	2-14
2.6.4	<code>ioi_intr</code>	2-14
2.6.5	<code>ioi_sio</code>	2-15
2.6.6	<code>ioi_hio</code>	2-16
2.6.7	<code>ioi_stpio</code>	2-16
2.6.8	<code>ioi_rsctl</code>	2-17
2.6.9	<code>ioi_wio</code>	2-17

2.6.10	ioi_seterr	2-17
2.7	Device Driver Entry Points	2-18
2.7.1	<i>dev_init</i>	2-18
2.7.2	<i>dev_intr</i>	2-19
2.7.3	<i>dev_maint</i>	2-20
3	IOI Support for Class E Devices	3-1
3.1	Overview	3-1
3.2	Class E I/O	3-1
3.3	I/O Interface Modifications	3-2
3.3.1	Extensions to Existing Routines	3-2
3.3.2	New Routines	3-3
	<i>ioi_sio_e</i>	3-3
	<i>ioi_hio_e</i>	3-4
	<i>ioi_phys_e</i> and <i>ioi_physr_e</i>	3-4
	<i>ioi_freemem</i>	3-5
	<i>ioi_rsctl_e</i>	3-5
	<i>ioi_cd_e</i>	3-6
	<i>ioi_td_e</i>	3-6
	<i>ioi_wio_e</i>	3-7
	<i>ioi_ictl_e</i>	3-7
	<i>ioi_ei_e</i>	3-7
	<i>ioi_di_e</i>	3-8
3.3.3	Data Structures	3-8
4	The Generic HSD Driver	4-1
4.1	Overview	4-1
4.2	HSD Data Structures	4-1
4.3	The Generic Driver	4-3
4.4	Customization Issues	4-3
4.4.1	Overview of Issues	4-3
4.4.2	Buffering Approaches	4-4
	Reading and Writing in User Space	4-4
	Using Kernel Buffers	4-5
4.4.3	Blocking and Nonblocking I/O	4-6
	Managing Command Queues	4-6
	Using a State Machine Model	4-7
	Handling Sleeps and Wakeups	4-8
	Checking for I/O Completions	4-8
4.4.4	Command Semantics	4-9
	<i>ioctl</i> Semantics	4-9
	Network Device Semantics	4-9
4.4.5	Shared and Exclusive Device Access	4-10
4.4.6	Configuring the Device	4-11

5	Direct Input/Output	5-1
5.1	Overview	5-1
5.2	Application	5-1
5.3	Issuing I/O Commands	5-2
5.3.1	Virtual to Physical Address Translation	5-2
5.3.2	Ensuring Physical Contiguity	5-2
5.3.3	Disabling Kernel Checking	5-2
5.4	Priority Ordering	5-3
5.5	I/O Request Tracking	5-3
5.5.1	Identifying Outstanding Requests	5-3
5.5.2	Notification of I/O Completions	5-3
5.6	Connecting and Disconnecting	5-4
5.7	Reserving a Device	5-4
	Appendix A Input/Output Interface Specification Files.....	A-1
A.1	Sample Device Driver Interface	A-1
A.2	sel/selio.h	A-3
A.3	selio/loi.h	A-6
	References	RF-1

Figures

Figure		Page
2-1	Location of the IOI in the Kernel	2-1
2-2	Data Structure and IOI Execution Paths	2-8
3-1	Class E Data Structures	3-2
4-1	HSD Data Structures	4-1
4-2	HSD Status and Error Return Structures	4-2
4-3	State Transition Diagram for the Generic HSD Driver	4-7

1 Introduction

This introductory chapter provides the following information about this guide:

- Its scope and purpose
- A summary of its contents
- Reader prerequisites
- Typographic conventions

1.1 Scope and Purpose of this Guide

This guide presents detailed information on the input/output (I/O) subsystem for UTX/32™ including its real-time extensions and enhancements. The I/O subsystem contains two major parts: the *system configuration utility* and *input/output interface* (IOI). The system configuration utility allows sites without source licenses to add custom device drivers to the system or to tune system parameters. For more information on the configuration utility, refer to the *UTX/32 Operations Guide*. The IOI, a portion of the UTX/32 kernel, provides an interface between the UTX/32 device drivers and the devices themselves.

1.2 Summary of Contents

This guide is divided into five chapters, an appendix, and a reference list.

Chapter 1	Provides general information about this guide
Chapter 2	Is an overview of the IOI, focusing on its general-purpose (non-real-time) features
Chapter 3	Contains the basics of class E I/O and descriptions of the IOI features that support it
Chapter 4	Is an overview of the generic, extensible device driver for the high-speed data interface (HSD), a commonly used class E device, and an example of how to customize it
Chapter 5	Explains how to use the direct I/O (DIO) facility, which provides low overhead I/O services to real-time processes
Appendix A	Contains samples of IOI specification files
References	Contains full citations of the non-UTX/32 documents referred to in this guide

You need not read the guide in the above order. Chapter 2 is an independent module that can be read separately. Chapters 3 through 5 provide information specific to real-time I/O.

1.3 Reader Prerequisites

Readers should be familiar with UTX/32 and to have access to its documentation. See the *UTX/32 Software Release Notes* and the *UTX/32 Documentation Guide*.

1.4 Typographic Conventions

The typographic conventions for this guide are described below.

The following prompt is used in this guide:

\$ Bourne shell prompt

Nonprinting and control characters

Nonprinting characters obtained by striking special keys are displayed within angle brackets. For example, indicates the delete key, <CR> a carriage return.

In this guide, a <CR> is assumed at the end of every command line unless otherwise stated. The <CR> is displayed only if nothing else is entered on the line or if the sequence of keystrokes would otherwise be unclear.

Control characters are represented using the caret notation. For example, ^D indicates <CTRL>-d. In examples, control characters are shown as echoing on the terminal screen. Whether they echo on your terminal depends on its settings; see *stty(1)*.

Boldface

Command and utility names, filenames, pathnames, and words from code are printed in boldface.

Example:

The **nroff** command is used to format text.

Exception: When such a term is long and all uppercase, such as PLOCK_FRACTION, it is not printed in boldface.

Lineprinter and **lineprinter bold**

Displays of code and user sessions are printed in lineprinter font. In displays of interactive user sessions, text typed by the user is printed in lineprinter bold.

Example:

```
$ ls
  file1      file2      file3
```

Italics

Variable expressions that must be replaced with a value are printed in italics. Square brackets ([]) around an italicized variable expression signify that specifying the value is optional.

Example:

```
% cd [directory]
```

Italics are also used to introduce new terms, for titles of documents or manual pages, and occasionally for emphasis.

Examples:

See *mount*(8) for further information.

The first tape, called the *boot tape*, contains three boot programs.

Blank pages

Since each major section of the document begins on a right-hand (odd-numbered) page, blank left-hand (even-numbered) pages occasionally precede new sections. You can be assured that such a page is intended to be blank if the preceding page has a double page number, such as 4-5/4-6.

Manual pages

References to manual pages with manual section specifiers ending in RT such as *dioconnect*(3RT) refer to real-time-specific manual pages in the *UTX/32 BSD Programmer's Reference Manual*. If there is also a FORTRAN version of the manual page, it will have a section specifier ending in RF. A reference such as *dioconnect*(3RT/RF) indicates that there are two manual pages, *dioconnect*(3RT) and *dioconnect*(3RF).

2 The Input/Output Interface

This chapter describes the input/output interface (IOI) of the Gould UTX/32 kernel for general-purpose features. Information provided includes:

- Overview of the IOI
- IOI external interfaces for device drivers, hardware devices, and the maintenance program
- Function of the IOI at system initialization
- Data structures required by the IOI
- Description of IOI execution paths
- IOI entry points through calls from the device driver and through hardware interrupts

For information on IOI support for class E devices, see Chapter 3.

2.1 Introduction

Figure 2-1 illustrates the IOI as an interface between the device drivers and the devices themselves. The IOI is shared by the device drivers and performs

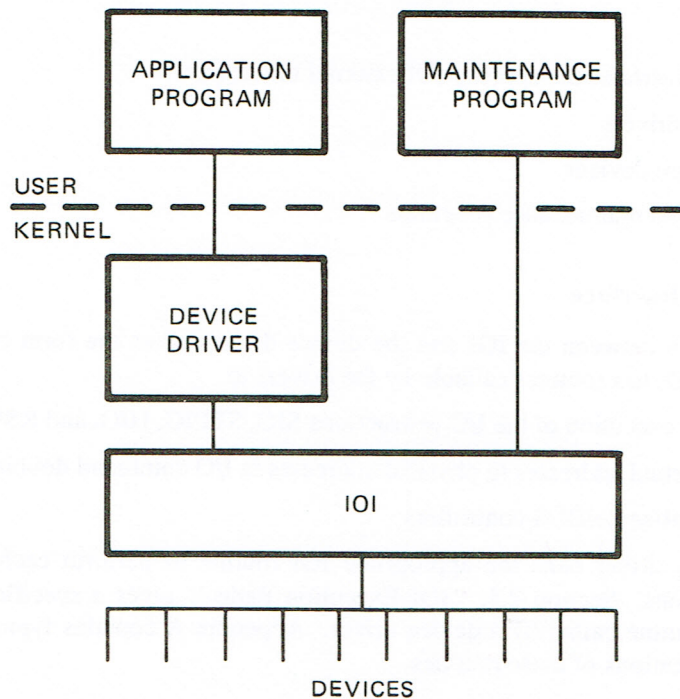


Figure 2-1. Location of the IOI in the Kernel

functions common to more than one device driver. Specifically, the IOI provides the device drivers with the following services:

- Execution of I/O instructions such as start I/O (SIO), stop I/O (STPIO), halt I/O (HIO), and reset controller (RSCTL)
- Fielding and routing of interrupts
- Virtual address to physical address mapping
- Optional error-recovery assistance
- Reinitialization of SelBUS™ controllers

In addition, the IOI directs the initialization of the I/O system during system initialization and provides a maintenance interface to privileged user processes. The manner in which these services are provided is discussed in the following sections.

The IOI extends the idea of moving certain error-recovery operations (such as issuing a sense-status channel program) into a separate section of code. This reduces the number of states required by the driver for error-recovery. It also allows for reasonable processing in situations such as the obtaining of a unit check during a sense-status operation. The IOI does this by providing error-recovery and similar services to all drivers. The design of the IOI allows drivers to specify whether address mapping or error-recovery assistance is needed from the IOI.

2.2 Interfaces

Figure 2-1 illustrates the specific IOI external interfaces:

- Device drivers
- Hardware devices
- Privileged maintenance programs

2.2.1 Device Driver Interface

The interface between the IOI and the device drivers takes the form of routine calls. The IOI has routines callable by the drivers to

- Initiate execution of the I/O instructions SIO, STPIO, HIO, and RSCTL
- Map virtual addresses to physical addresses in I/O command doublewords
- Reinitialize SelBUS controllers

Each device driver calls the appropriate IOI routine to perform each of these three functions. Section 2.5, "IOI Execution Paths," gives a specification for each IOI routine called by a device driver. Appendix A contains typical device driver applications of these routines.

Each device driver must provide routines for the IOI to call. The IOI calls the following routines:

- *dev_init()* to initialize the driver during system initialization
- *dev_intr()* to process interrupts for that device
- *dev_maint()* to request or release a subchannel on behalf of a privileged user process

where *dev* represents any one of a series of abbreviations for standard device drivers (see Section 2.3.1, "Logging"). The device driver entry points are specified in Section 2.7, "Device Driver Entry Points."

2.2.2 Device Interface

All interaction with devices occurs in the device interface. At the request of the device drivers, the IOI uses data structures created during system configuration/generation to send information to the devices by executing extended I/O instructions. The IOI receives information from the devices through condition codes set by the I/O instructions and through interrupts. Condition codes are sent to the drivers as return codes from the routine calls issuing the I/O instructions. Interrupt information is passed to the drivers through calls to the drivers' interrupt service routines.

Device status may also be presented to the IOI when an I/O instruction is executed. When this occurs, the new status is presented to the driver's interrupt service routine. If the subchannel presenting status is equal to the subchannel to which the I/O instruction referred, the instruction is not retried. Instead, the caller is informed that an interrupt is pending on that subchannel.

2.2.3 Maintenance Interface

A maintenance interface is provided that allows a privileged user process to execute channel programs and other I/O commands for configured devices. Additionally, a privileged user process may communicate with drivers that provide an entry point for the purpose. The interface for executing I/O instructions resides in the IOI, providing a consistent interface from device to device, simplifying what each device driver must do to allow maintenance access to its devices.

The following occurs when a privileged process requires direct access to a device:

1. The process opens */dev/ioi_n*, where *n* is a value 0 - 3.
2. The process fills in the *maint_req* data structure with the channel address of the device to be accessed (see Appendix A, Section A.2).
3. After *maint_req* is built, the process calls the *ioctl()* routine with a pointer to *maint_req*. The *ioctl()* operation code (opcode) used for this call is *IO_REQUEST*. This call allows the process to request a device to be placed in maintenance mode.

4. The IOI receives **maint_req** and determines if there is a device control block (DCB) for the device. If not, the value -1 is returned and the external variable **errno** contains the return value **ENODEV**.
5. The IOI calls the device maintenance interface routine *dev_maint*. The first parameter is the address of the DCB for the device. The second parameter indicates that the request is for access to the device **MR_REQUEST**. The device maintenance routine can also be called with the second parameter set to values **MR_RELEASE** and **MR_COMM**, which will be described later.
6. The device driver either approves or denies the request. If the driver cannot relinquish the device (for example, because it is in use), it returns the value **EBUSY**. If the driver can relinquish the device, it notes internally that the device is in use for maintenance and returns the value zero.
7. The IOI passes the result of the call back to the maintenance process through the external variable **u.u_error**. A zero return value to the user process indicates success. A -1 return value indicates failure, and the reason is found in **errno** (see *errno(3)* and *intro(2)*).
8. The user process executes one device reservation **IO_REQUEST ioctl()** for each subdevice needed to be simultaneously accessed.
9. The user process is now ready to perform I/O. The user process builds the **maint_rep** data structure that describes the operation to be supported. Procedures **ioi_sio()**, **ioi_hio()**, **ioi_stpio()** or **ioi_ictl()** may be accessed by using the following opcodes with **ioctl()**: **IO_SIO**, **IO_HIO**, **IO_STPIO** or **IO_IOCTL**, respectively. Additional fields provide arguments to the call. For definitions of the fields, see the data structure declarations in Appendix A, Section A.3.

After the data structure is built, it is passed to the IOI by the **ioctl()** system call with one of the opcodes specified above. In the event of error, **errno** contains a value indicating the reason for failure. The call to **ioctl()** does not block, so several channel programs may be executed simultaneously by executing separate **ioctl()** calls requesting **ioi_sio()** calls on different subchannels.

10. The user process now waits for an interrupt by executing an **ioctl()** system call with **IO_WAIT** as the opcode. This call blocks until an event related to one of the reserved DCBs occurs. When such an event occurs, the IOI completes the system call by filling in the user-specified memory area with a copy of the DCB, and with the **maint_result** data structure containing the reason the **ioctl()** completed. The reason field of **maint_result** takes on the same values as the **reason** parameter for a device driver's interrupt service routine.
11. The user process now takes the appropriate action such as building and executing another channel program.

12. When the user process is finished with a subchannel, it may either explicitly release the subchannel using an `ioctl()` call (with opcode `IO_RELEASE`), or implicitly release all DCBs allocated to it by closing the `/dev/ioin` file.
13. For each DCB (that is, subchannel) released, the device maintenance routine `dev_maint` is called. The first parameter is a pointer to the DCB describing the subchannel being released. The second parameter indicates that the DCB is being released (opcode `MR_RELEASE`).

The privileged maintenance process may use the `ioctl()` opcode `IO_COMM` to communicate with drivers whose maintenance routines are written to support this communication. Any channel address belonging to the driver may be used, and the DCB associated with the channel address does not have to be reserved by the maintenance process. The second parameter to the maintenance process is opcode `MR_COMM`; the third parameter is the address of the maintenance process `comm` buffer. The manner in which the driver and the maintenance process use `comm` is device driver-dependent; not all drivers support this opcode.

Directory `/usr/lib/libio.a` includes a library of routines to make some of the previously mentioned actions easier to perform (see `ioi(7)`). If you have a source software license, you may want to examine the SCM initialization daemon and device driver.

2.2.4 System Configuration Interface

The IOI uses data structures and assembly language routines produced during system configuration/generation. The content and use of these structures are discussed in Section 2.4, "Data Structures." For the declarations of these class F I/O structures, refer to Appendix A, Section A.2.

2.3 System Initialization

At system initialization, the kernel initialization routine calls the `ioi_init()` routine. `ioi_init()` initializes the SelBUS controllers, Multi-Function Processors (MFPs), and IOPs, and calls each device initialization routine. See Section 2.6, "IOI Entry Points," for more information on `ioi_init()`.

2.3.1 Logging

The IOI provides a debug logging facility. It consists of a circular buffer and routines that do the following in the buffer:

- Make entries
- Print entries

- Enable the logging of entries
- Disable the logging of entries

As distributed, the kernel is compiled without the logging option. Source software licensees can set the makefile variable `DDOPTS` to

```
DDOPTS=-DQQDEBUG
```

and recompile all I/O modules.

The IOI and device drivers must have access to logging for debugging and hardware diagnostic purposes.

The routine that makes entries in the buffer is called `qqlog()`. By convention, `qqlog()` is invoked only by the define `QQLOG()`. `QQLOG()` accepts a variable number of arguments of the format:

```
QQLOG((routine, reason, nargs [, arg ...]))
```

NOTE: The extra set of parentheses is required. *nargs* may be a value 0 - 4 and should be followed by the appropriate arguments. Values for *routine* and *reason* may be selected from file `selio/ioi.h`. Additional entries may be added. For convenience, the predefined `QQLOG LOG_ENTRY reason` and `LOG_EXIT reason` may be used on entry to and exit from a routine to delimit the routine's log activity.

The routine that prints the buffer is `qqdump()`. It is declared in file `selio/i_logging.c`. Tables of strings corresponding to the defined routine names and reasons passed to `QQLOG()`, allow `qqdump()` to decode the circular buffer into legible text. After a system crash, `qqdump()` may be invoked by setting the program counter to the value of the symbol `doqqdump()` and placing the processor in the RUN state.

Logging to the circular buffer may be disabled or reenabled by calling `qqoff()` or `qqon()`, respectively.

For hardware diagnostic purposes, the console `printf()` is used. The kernel keeps the last 8-Kbyte characters printed to the console in a ring buffer. This buffer is periodically read by a user process and written to a disk file. Later, the disk file can be searched using `grep` or `vi` to locate error messages for specific devices. Device drivers calling `printf()` use the following format in order to make the searches successful:

```
dev: message
```

Standard device drivers use one of the following values for *dev*:

```
as      8-Line Asynchronous Communications Multiplexer (8-Line Async)
cn      Console
```

dk	Disk
en	Ethernet™
fl	Floppy disk drive
lp	Lineprinter
md	Memory disk
mt	Magnetic tape
sc	Synchronous Communications Multiplexer (SCM)
sd	SCSI disk
st	SCSI tape

Custom device drivers should use the same two-letter identifier used in their configuration file entry. The string **ioi:** precedes errors logged by the IOI.

2.4 Data Structures

The following data structures are produced and initialized during system configuration/generation. They are required by the IOI.

SCR	Device and interrupt entries portion of the processor scratchpad
SIV	Service interrupt vector for SelBUS interrupts
ICB	Interrupt context block
DCB	Device control block
DPT	Device parameter table
SRT	Subchannel interrupt routing table
MTD	Minor-device-to-DCB mapping table

Figure 2-2 illustrates the relationships of the data structures discussed in the following sections. See Appendix A, Section A.2, for the declarations of the class F I/O structures.

2.4.1 Processor Scratchpad (SCR) Entries

The device address and interrupt priority information in the scratchpad is fundamental to starting I/O operations and to servicing I/O interrupts. The system configuration process generates 128 device and 112 interrupt entry initializations. These are merged by kernel code with software-defined interrupt entries and stored in the scratchpad at initialization time.

The processor indexes into the device or interrupt entries during an I/O operation or an interrupt operation, respectively. During interrupt operations, the device priority is used to index the service interrupt vector (SIV), which in turn is used to find the ICB for the interrupting device.

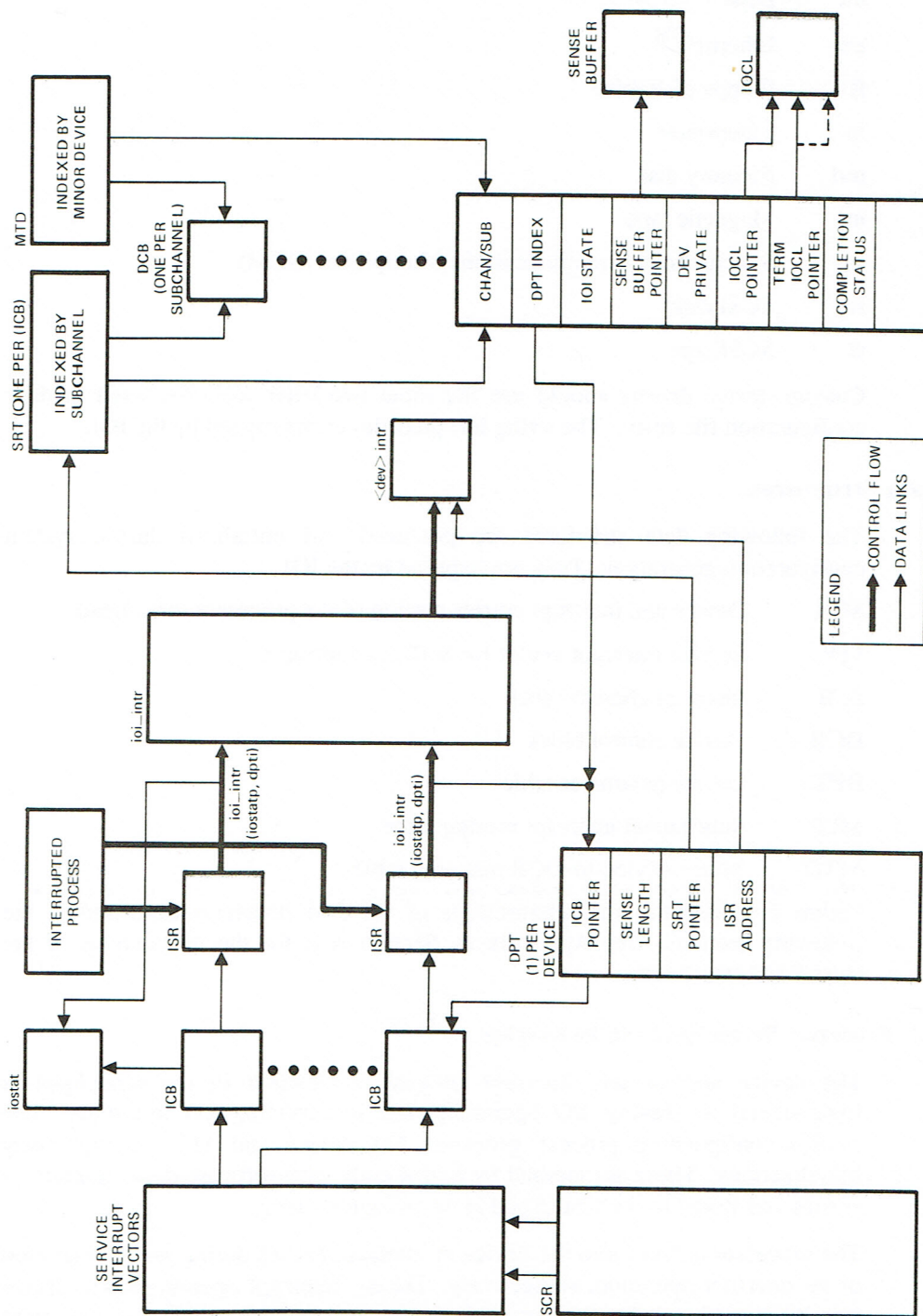


Figure 2-2. Data Structure and IOI Execution Paths

2.4.2 Service Interrupt Vector (SIV)

The SIV consists of a list of pointers to the ICBs. It is accessed by the processor to locate the ICB for an interrupting device. The device priority (obtained from the device entry in the scratchpad) is used to index the SIV. The address of the SIV is stored in the scratchpad by the kernel initialization code. During system initialization, the kernel merges the SIV with software-defined interrupt entries and loads them into the scratchpad.

2.4.3 Interrupt Context Block (ICB)

The hardware requires an ICB for each SelBUS device that may cause an interrupt. The ICB is used to save the old context when an interrupt or trap occurs, and contains the new context for servicing the interrupt.

Each ICB associated with a class F I/O device contains a location where the processor stores a pointer to the status word provided by the interrupting controller. Each ICB for a class F I/O device also contains a word used to point to the I/O command list for start I/O operations.

The IOI uses only ICBs involved in I/O operations.

2.4.4 Device Control Block (DCB)

The DCB is the central data structure for controlling I/O on a specific subchannel. One structure is built for each subchannel configured. The structure contains the following:

Channel/subchannel address

Read-only by the IOI and the device driver

This address is set during system configuration.

Minor device number

Read-only by the IOI and the device driver

This number is set during system configuration.

Device parameter table (DPT) index

Read-only by the IOI and the device driver

This index is set during system configuration.

IOI-state vector

For private use of the IOI

Sense buffer pointer

Read/write by the IOI; read-only by the device driver

The sense buffer pointer points to a sense buffer allocated by the IOI. The length of the allocated sense buffer is specified by the device's DPT entry. This buffer is read/write by the IOI and read-only by the device driver.

Sense count

Read/write by the IOI; read-only by the device driver

The sense count is set by the IOI during error-recovery to indicate how many bytes of data are in the sense buffer. This count is intended to accommodate devices that may return variable amounts of sense data. The count never exceeds the maximum sense count for the device. A count of zero indicates sense data is not present. The IOI may deallocate the sense buffer after the device interrupt service routine returns control to the IOI.

IOCL pointer

Read/write by the IOI; read-only by the device driver

The IOCL pointer points to the current I/O command list for the subchannel. It is set by the IOI when `ioi_sio()` is called. The driver may use this value during interrupt servicing for error-recovery or for deallocating the IOCL after the successful completion of the I/O.

I/O completion status

Read/write by the IOI; read-only by the device driver

The I/O completion status contains the channel status doubleword pointed to by the ICB when the interrupt occurred. The fields are as follows:

- Subaddress of completed I/O
- Pointer to terminating IOCD + 8 bytes
- Channel and device status flags
- Residual byte count

Normally, the driver uses only the last three fields of the structure.

Exclusive use word

This is read/write by the driver and is for the exclusive use of the driver. This word may be used any way the driver needs. The IOI never touches it.

2.4.5 Device Parameter Table (DPT)

A DPT is a structure containing a list of device parameters. There is one entry for each SelBUS controller, each IOP, and each IOP-based controller or MFP. Each entry includes the following parameters:

- ICB pointer
- Sense length
- Driver initialization routine address
- Driver interrupt service routine address
- Driver maintenance routine address

- Number of subchannels (subchannel interrupt routing table length)
- Pointer to the subchannel interrupt routing table
- Type of controller
- Address of controller working storage
- Pointer to a character string containing the name of the device

2.4.6 Subchannel Interrupt Routing Table (SRT)

Entries in the SRT map subchannel addresses to DCB addresses. The `ioi_intr()` routine finds SRTs through the DPT entry for the interrupting device. `ioi_intr()` indexes the SRT by the subchannel address to locate the DCB associated with the subchannel. There is one SRT for each SelBUS controller and each IOP or MFP.

2.4.7 Minor-device-to-DCB Mapping Table (MTD)

One MTD is built for each device driver. Each table has one entry per subchannel. The tables are used by the device drivers to map minor device numbers into DCB addresses. If the device type associates more than one subchannel with a physical device, the subchannels are grouped together in the MTD. Then the driver must index the MTD by the minor device number times the number of subchannels per device.

If the minor device address space is not contiguous, the configuration program `/etc/config` fills the holes in the address space with null entries. These null entries appear in the table to reflect the holes. If the IOI initialization routines discover that a controller is not present on the SelBUS, the IOI locates and changes the proper MTD entries to zeros.

Drivers are responsible for refusing open requests on devices having null MTD entries. Additionally, drivers such as disk or tape must issue a sense channel program during their `open()` routine before completing the open request, to ensure that there are mounted media.

2.5 IOI Execution Paths

Two major execution paths exist through the IOI (see Figure 2-2). One path starts with calls from the device driver. The other path starts with device interrupts. The following two sections discuss the paths.

2.5.1 Calls from the Device Driver

Driver routines call IOI service routines directly. All IOI routines that perform I/O require a pointer to the DCB for that subchannel. Device driver routines such as `dev_read` and `dev_write` obtain this pointer by using the driver's MTD array. The device driver interrupt routine `dev_intr` does not reference the MTD array because when called by the IOI, the routine receives the address of the DCB.

The IOI service routines store state information in the DCB for the referenced subchannel, and execute the requested I/O instruction. The SIO instruction requires that the address of the channel program be stored in the ICB for that controller. The ICB is found by indexing the device parameter DPT with the DPT index from the DCB.

2.5.2 Device Interrupts

The following flow of control is illustrated in Figure 2-2.

1. When a hardware interrupt occurs, the processor finds the correct location in the SIV by means of the entries in the hardware scratchpad SCR.
2. The addressed SIV location points to the ICB for the particular SelBUS device.
3. The old processor status doubleword (PSD) and a pointer to the channel status structure are stored in the ICB.
4. The new PSD is fetched from the ICB and loaded.
5. The new PSD points to an assembly language routine (*dev_isr*) generated by the system configuration process. This routine builds a new stack frame on its own private stack, pushes the channel status doubleword address found in the ICB and DPT index onto the stack, and calls *ioi_intr()*.
6. *ioi_intr()* accesses the channel status doubleword pointed to by the first parameter to determine the subchannel address, indexes the DPT to determine the SRT address, and then indexes the SRT with the subchannel address to locate the DCB. *ioi_intr()* uses the channel status information pointed to by the ICB, plus the IOI-state information in the DCB, to determine whether to perform additional error-recovery or to call *dev_intr*. *ioi_intr()* obtains the driver's *dev_intr* address from the DPT.

Interrupts from devices for which there is no DCB are considered erroneous and produce an error message on the console.

2.6 IOI Entry Points

The IOI may be entered through

- Calls from the device driver
- Hardware interrupts

The following subsections describe IOI entry points.

2.6.1 ioi_init

Call format	ioi_init()
Purpose	ioi_init() is called by the system at system initialization. It initializes the various SelBUS controllers, IOPs or MFPs, and calls each driver initialization routine. ioi_init() clears the MTD entries associated with controllers found to be offline during system initialization.
Parameters	None.
Return value	None.

2.6.2 ioi_phys

Call format	ioi_phys(proto_ioclp, dest_ioclp, dest_len, procp)
Purpose	ioi_phys() performs memory mapping functions for I/O devices. This operation is performed by expanding channel command words containing virtual addresses into channel command words containing physical addresses. Data chaining is used if the buffer described by the virtual address(es), and the count(s) in the prototype I/O command list proto_ioclp cross a page boundary. ioi_phys() returns a value indicating whether the conversion was successful or why it was not. If the caller uses dynamically allocated memory, the caller must later deallocate that memory, as the IOI does not deallocate it. ioi_phys() calls pvtophys(addr, procp, direction) to convert virtual addresses to physical addresses. By using a process table pointer, it is possible to perform mapping for processes in memory, even when called from an interrupt service routine.
Parameters	proto_ioclp(*iocdT) is a pointer to the prototype I/O command list to be converted from virtual to physical addressing mode. dest_ioclp(*iocdT) is a pointer to the destination area for the converted I/O command list. If it is zero, memory is allocated for it. dest_len(int) is the maximum number of I/O command lists of the destination area for the converted I/O command list.

procp(struct proc*) is a pointer to the process table entry for the process requesting the I/O. It should be set to zero for requests involving kernel virtual addresses.

Return value

Zero indicates the conversion was successful.

EFAULT indicates that some part of the specified user buffer is outside the virtual address range assigned to the user.

EINVAL indicates that the channel program to be converted is too long or contains an illegal opcode such as transfer in channel (TIC).

2.6.3 ioi_ictl

Call format

ioi_ictl(dpti)

Purpose

ioi_ictl() reinitializes a SelBUS controller on behalf of a SelBUS device driver. Currently, only disk initialization and tape controller initialization are supported. As a last resort, disk and tape drivers may call this routine in their error-recovery process to clear a hung controller.

This routine cannot be used to initialize a device that was found offline (with respect to the SelBUS) during system initialization, because the DCB pointers in the MTD with which the controller is associated have been permanently lost.

Parameters

dpti(int) is an index into the device parameter table. The index can be obtained from any DCB for the offending device.

Return value

None.

2.6.4 ioi_intr

Call format

ioi_intr(statp, dpti)

Purpose

ioi_intr() performs the interrupt service operations common to all device drivers. After performing the operations, **ioi_intr()** calls the interrupt service routine for the associated driver.

ioi_intr() is called by the assembly language interrupt service routines generated during system configuration. These routines set up the stack, push both a pointer to the ICB through which the interrupt occurred and the appropriate index to the device parameter table onto the stack, and call **ioi_intr()**.

Parameters **statp(*iostatT)** is a pointer to the channel status doubleword presented to the CPU and stored in the ICB when the hardware interrupt occurred.

dpti(int) is the index into the DPT. This is used to find the address and length of the appropriate SRT.

Return value None. However, **ioi_intr()** logs an error message to the console if an interrupt occurs for a subchannel that is not configured.

2.6.5 ioi_sio

Call format **ioi_sio(dcbp, startiocp, timeoutvalue, errprocflags)**

Purpose **ioi_sio()** starts I/O (that is, executes a channel program) on behalf of the caller. If the channel program fails, various error handling options are available. These options are discussed in the section entitled *dev_intr*.

Parameters **dcbp(*dcbT)** points to the DCB describing the subchannel on which the I/O is to be performed.

startiocp(*iocdT) points to the IOCL to be executed. The IOCDs in the IOCL must describe locations in physical memory. The driver may use **ioi_phys()** to convert IOCLs containing virtual addresses into IOCLs containing physical addresses.

timeoutvalue(int) may be set to a nonzero value if the caller wishes to have the channel program terminated by the IOI if the program does not complete within a specified number of seconds. In the event of a time out, the manner in which the I/O is terminated is determined by the value of the error processing flags **errprocflags**.

errprocflags(int) are error processing flags indicating the action the IOI must take when the I/O cannot be completed normally. Generally, the IOI obtains sense information for the driver if these flags are set correctly. Error processing flags perform the following functions:

IE_SUC	Obtains sense information on a unit check
IE_SUE	Obtains sense information on a unit exception
IE_SAT	Obtains sense information on an attention
IE_SIL	Obtains sense information on incorrect length

IE_RTYCBY Retries the operation if it is refused due to a busy controller

IE_HTO Issues an HIO on time out

IE_RTO Issues a reset-controller instruction on an HIO time out

Return value The return value from `ioi_sio()` indicates whether the I/O was started successfully.

IS_OK Indicates that I/O started successfully

IS_BY Indicates that I/O failed due to a busy controller

IS_IP Indicates that I/O failed due to an interrupt pending on this subchannel

IS_RTCNT Indicates that I/O failed due to a busy controller

IS_BADSTATE Indicates that the IOI did not attempt to start the I/O because the subchannel was in the wrong state

2.6.6 `ioi_hio`

Call format `ioi_hio(dcbp, timeoutvalue, errprocflags)`

Purpose An HIO has been issued to the specified subchannel.

Parameters The meanings of the parameters `dcbp`, `timeoutvalue`, `errprocflags`, and the return value are the same as for `ioi_sio()`, except that `IE_HTO` is a no-op.

Return value The return values are the same as for `ioi_sio()`.

2.6.7 `ioi_stpio`

Call format `ioi_stpio(dcbp, timeoutvalue, errprocflags)`

Purpose An STPIO instruction has been issued to the specified subchannel.

Parameters The meanings of parameters `dcbp`, `timeoutvalue`, `errprocflags`, and the return value are the same as for `ioi_sio()`, except that `IE_HTO` is a no-op.

Return value The return values are the same as for `ioi_sio()`.

2.6.8 ioi_rsctl

Call format	ioi_rsctl(dcbp, errprocflags)
Purpose	ioi_rsctl issues a reset-controller instruction to the subchannel indicated by the specified DCB.
Parameters	The meaning of dcbp is the same as for ioi_sio() . The only error-recovery bit defined in errprocflags for this call is RTY_CBY . All other error-recovery bits are no-ops.
Return value	The return values are the same as for ioi_sio() .

2.6.9 ioi_wio

Call format	ioi_wio(dcbp)
Purpose	ioi_wio() is called by the console routine when it must wait for an I/O operation to complete with the interrupts disabled. The routine must be called with interrupts disabled at, or higher than, the priority level of the I/O for which the ioi_wio() is waiting. The driver's interrupt routine is called when the I/O completes. After that, ioi_wio() returns to the caller.
Parameters	dcbp(*dcbT) points to the DCB that describes the subchannel on which the I/O is to be performed.
Return value	IS_OK Indicates successful completion of the I/O

2.6.10 ioi_seterr

Call format	ioi_seterr(dcbp, errprocflags)
Purpose	ioi_seterr() sets the error processing flags for spontaneous interrupts. It is generally called from the device init routine, but may be called at any time to set or reset the spontaneous error processing flags. If a driver does not call ioi_seterr , the IOI does not obtain sense information on a spontaneous interrupt.
Parameters	dcbp(*dcbT) points to the DCB describing the subchannel on which the error processing is to be performed. errprocflags(int) are error processing flags indicating the action the IOI must take when the I/O cannot be completed normally. Generally, the IOI obtains sense information for the driver if the flags are set correctly. Error processing flags perform the following functions:

IE_SUC	Obtains sense information on a unit check
IE_SUE	Obtains sense information on a unit exception
IE_SAT	Obtains sense information on an attention
IE_SIL	Obtains sense information on incorrect length

Return value None.

2.7 Device Driver Entry Points

There is a DCB data structure allocated during system configuration for each subchannel that a device driver handles. To enable a driver to find the DCBs for its devices, an MTD is built for each driver during system configuration. The MTD maps a minor device number to a DCB address.

In the following sections, the *dev* field is an abbreviation representing the standard device driver (see Section 2.3.1, "Logging").

2.7.1 *dev_init*

Call format *dev_init*()

Purpose *dev_init*() is called by the IOI at system initialization to allow the device driver to perform any necessary device-dependent initialization. Disk and tape drivers are not required to initialize their controllers. Drivers for IOP-based controllers requiring initialization (such as the SCM) may choose to initialize their controllers now or when they receive the first open request.

The DCB structure contains a word for the exclusive use of the device driver. At initialization, the driver may choose to link each DCB to its private data structure for that minor device. The DCBs may be found through the MTD. A global variable containing the length of the MTD is also available.

dev_init() may be called more than once during system initialization. The driver should keep a state variable to indicate whether its initialization has been completed. Redundant calls to *dev_init*() are treated as no-ops.

Parameters A parameter of zero value is currently passed. This parameter may be used later to indicate whether reinitialization is requested. This might occur, for instance, if a controller was connected to a reinitialized IOP. Presently, this parameter can be ignored.

Return value None.

2.7.2 *dev_intr*

Call format *dev_intr(dcbp, reason)*

Purpose This routine is the addressed device driver's ISR. It is responsible for determining the reason for the call and for taking the appropriate action. The **reason** parameter distinguishes among the following cases:

Successful I/O completion

In most cases, *dev_intr(dcbp)* is called when I/O successfully completes. In this case, the ISR notifies the user or system task that originally requested the driver to perform the I/O. The ISR starts the next I/O operation.

I/O completion with abnormal status

The interrupt fielded by the IOI had an abnormal status (that is, it was not equal to channel end/device end). If the error handling flags to *ioi_sio()* requested them, sense data are obtained. The ISR is then responsible for taking appropriate action. The availability of sense data is indicated by a nonzero sense buffer count in the DCB.

Spontaneous interrupt

Some controllers may interrupt spontaneously. The ISR may then take appropriate action. The IOI has already attempted to obtain sense information if Attention or Unit Check was set with the spontaneous interrupt.

I/O successfully halted by request or timeout

The executing channel program was cancelled by an HIO or STPIO instruction. This may be due to a call to *ioi_hio()* or *ioi_stpio()*, or to a timed out *ioi_sio()* request. The ISR performs the appropriate action. A typical use of this is in handling tandem flow control on TTY lines.

I/O cancelled by reset controller after **haltio** timeout

The executing channel program was cancelled by an RSCTL instruction. The ISR takes whatever action is appropriate. The ISR is not notified if the RSCTL was issued in response to a call to *ioi_rsctl()* because in this case, it is assumed that the driver knows the RSCTL has been executed.

Broken

The device failed to accept an HIO or RSCTL operation and has been declared broken by the IOI, or the command timed out and **halt** was not requested.

Parameters

dcbp(*dcbT) is a pointer to the DCB associated with the subchannel. From the DCB, the driver can determine such information as the reason for the interrupt, the presence of a sense buffer, and the terminating address of the IOCL. The terminating address may be useful for error-recovery on devices such as the disk. See Section 2.4.4, "Device Control Block (DCB)," for more information.

reason(int) is a code indicating the reason the driver's interrupt service routine is being called. **reason** may have one of the following values:

- ICS_OK Successful I/O completion
- ICS_AB I/O completion with abnormal status
- ICS_SI Spontaneous interrupt
- ICS_HIO I/O successfully halted by request or time out
- ICS_RSCTL I/O cancelled by reset controller after **haltio** time out
- ICS_BROKEN The device failed to accept an HIO or RSCTL operation and has been declared broken by the IOI, or the command timed out and **halt** was not requested

Return value

None.

2.7.3 dev_maint

Call format

dev_maint(dcbp, reason, id)

Purpose

dev_maint is called by the IOI when a privileged user process tries to access a subchannel controlled by this device driver.

Parameters

dcbp(*dcbT) points to the DCB for the requested subchannel.

reason(int) indicates the reason for the call and takes the following values:

MR_REQUEST can be used when a privileged process tries to access (and have exclusive use of) the subchannel described by the DCB pointer. If the driver accepts the request, the driver must set a state variable inhibiting the driver from further access to the device until it is released. The driver may refuse open requests for the device during this time.

MR_RELEASE can be used when the privileged process which controls the subchannel described by the DCB pointer is relinquishing control of the subchannel to the device driver. Note that some physical devices are described by more than one subchannel. The release of one subchannel does not imply the release of the device. The driver must wait until all subchannels for a physical device are released before accessing that device.

MR_COMM allows a buffer of data to be passed from the privileged process to the driver's maintenance interface routine. **MR_COMM** is currently used only for informing the SCM device driver of an SCM download.

dev_maint should set **u.u_error** to **EINVAL** for **reason** values it does not understand.

id(int) uniquely identifies the requesting process. The device driver may choose to use this ID to prevent allocating two DCBs describing a device to two processes.

Return value

Zero indicates the request was accepted. The subchannel now belongs to the privileged user process. The device driver does not access this subchannel (or other subchannels directly associated with this physical device) until it has been released.

EBUSY indicates the driver cannot allow access to the device at this time. Generally, this means that the device is already open.

3 IOI Support for Class E Devices

3.1 Overview

This chapter provides information on class E data structures. It then describes UTX/32 modifications to the I/O interface for class E I/O.

UTX/32 supports class E I/O devices. Most standard UTX/32 devices use class F I/O. Compared to class F I/O, class E I/O is a simple instruction set that requires more bookkeeping by the software that uses it. However, class E I/O permits specific devices to define more complex protocols and capabilities on top of its simple model. This simplicity and adaptability make class E devices appropriate for real-time applications.

UTX/32 includes a generic, customizable class E device driver for the high-speed data interface (HSD), a SelBUS controller that is a common class E device.

In this chapter, reference is made to IOCBs. IOCBs are I/O control blocks, which are HSD-specific I/O control list (IOCL) elements. See Chapter 4, "The Generic HSD Driver," for a discussion of HSD data structures and the generic HSD driver.

3.2 Class E I/O

Class E I/O uses only the command device (CD) and test device (TD) instructions. A CD instruction notifies the device that a command has been placed in a standard place and that the device should begin executing the command. A TD instruction gets the status of the device. Basic class E I/O operations and the data structures that support them are summarized in this section.

The following data structures underlie class E I/O:

- TAW Transfer address word, sometimes referred to as transfer control word (TCW). This structure contains an address interpreted by the class E device as a pointer to data or to operations to be performed.
- IOCD I/O control doubleword. This structure contains a pointer to the TAW and passes device-dependent commands to the device.

The IOCD and TAW are illustrated in Figure 3-1. Both structures are located in device-specific locations in memory. The first word of the IOCD contains the right half of the CD instruction, which has been loaded by the firmware. The second word contains the TAW address, which must be loaded by software. The TAW can contain either a definition (address and count, as shown) of data to be transferred, or a pointer to a more detailed definition of the operations to be performed. The interpretation is made by the device.

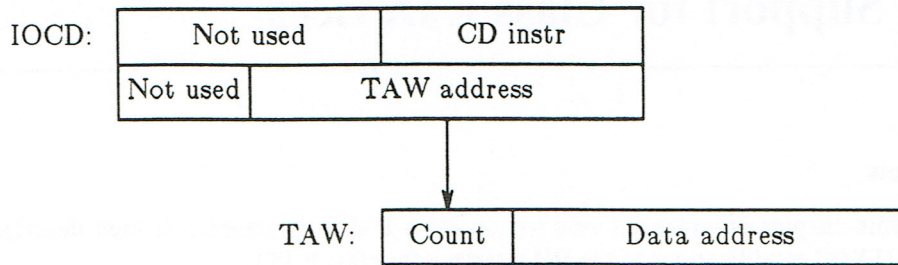


Figure 3-1. Class E Data Structures

Memory has been reserved in the kernel for both IOCDs and TAWs. The location of this memory can be found in the definition of `E_EMUL_IOCDs` and `E_TAWs` in `/usr/include/sel/selio_e.h`. There are 16 words reserved for class E TAWs, one word for each possible class E interrupt level. There are 32 words reserved for class E IOCDs, two words for each possible class E interrupt level. This reserved memory resides in the low 64K bytes of memory. Note that the class E IOCD's base address can also be found in the processor scratchpad.

3.3 I/O Interface Modifications

The I/O interface (IOI) is a collection of I/O support routines in the UTX/32 kernel. These IOI services include the following generic operations for class F devices:

- Virtual to physical IOCL address mapping
- Layered operations in which the IOI does the generic part of the operation and then calls a device-specific routine
- Device and controller initialization
- Interrupt dispatching

Class E devices require somewhat different data structures and interrupt handling. Therefore, UTX/32 extends the standard UTX/32 IOI to accommodate class E devices. The following sections describe these extensions.

3.3.1 Extensions to Existing Routines

UTX/32 extends certain existing class F UTX/32 IOI routines. The following is a list of the extended routines and extensions.

- `ioi_sio` Panics for class E devices.
- `ioi_hio` Panics for class E devices.

ioi_init	Calls ioi_ictl_e to initialize class E devices.
ioi_timeout	If a time out occurs with the device busy, does halt I/O with ioi_hio_e .

3.3.2 New Routines

UTX/32 adds the following IOI routines specific to class E devices:

ioi_sio_e	Starts I/O.
ioi_hio_e	Halts I/O.
ioi_phys_e	Performs memory mapping.
ioi_physr_e	Performs memory mapping in maintenance mode.
ioi_freemem	Unlocks memory locked by ioi_physr_e .
ioi_rsctl_e	Issues a reset controller and idles the affected devices.
ioi_ictl_e	Initializes an HSD driver.
ioi_cd_e	Issues a CD instruction.
ioi_td_e	Issues a TD instruction.
ioi_wio_e	Polls for I/O completion.
ioi_intr_e	Performs common interrupt services.
ioi_ei_e	Enables interrupts.
ioi_di_e	Disables interrupts.

The following sections summarize the call format, purpose, parameters, and return value for each of these routines. The routines are called by the device driver. All return values are of type **int**.

ioi_sio_e

Call format:

ioi_sio_e(dcbp, startioclp, timeoutvalue, errprocflags)

Purpose:

Issues a start IO (SIO) on behalf of a class E device.

Parameters:

dcbp	Type dcbT* . Device control block pointer.
startioclp	Type iocbT* . Pointer to queue of available IOCBs.
timeoutvalue	Type int . This may be set to a nonzero value if the caller wants the IOI to terminate the request upon time out.

errprocflags Type **int**. These flags control error-recovery:
IE_HTO Issues a halt I/O (HIO) on time out.
IE_RTO Issues a reset controller on HIO time out.

Return value:

IS_OK Indicates that I/O was started successfully.
IS_BY Indicates that I/O failed due to a busy controller.
IS_IP Indicates that I/O failed due to a pending interrupt.
IS_BADSTATE Indicates that the device is in the wrong state.

ioi_hio_e

Call format:

ioi_hio_e(dcbp, timeoutvalue, errprocflags)

Purpose:

Issues a halt I/O to a specified class E device.

Parameters:

dcbp Type **dcbT***. Device control block pointer.
timeoutvalue Type **int**. The time out value, if nonzero.
errprocflags Type **int**. The error processing that should be done.

Return value:

IS_OK The request was accepted.
IS_BY The channel or subchannel is busy.
IS_IP An interrupt is pending.

ioi_phys_e and **ioi_physr_e**

Call format:

ioi_phys_e(sp, dp, dn, procp)
ioi_physr_e(sp, dp, dn, procp, im_mp, im_dsp)

Purpose:

Perform memory mapping functions for a class E device. **ioi_physr_e** is called in maintenance mode.

Parameters:

sp Type **iocbT***. Source virtual IOCL address.

dp	Type iocbT* . Destination physical IOCL address.
dn	Type int . Destination physical IOCL size (number of IOCBs).
procp	Type procT* . Process pointer.
im_mp	Type ioi_mstateT* . Maintenance process table address.
im_dsp	Type im_dcbstateT* . Maintenance device control block state table address.

Return value:

0	Success.
EFAULT	Failure due to a memory fault.
EINVAL	IOCL too long or contains a transfer control block (not supported).

ioi_freemem

Call format:

ioi_freemem(l_iocl,n_iocls)

Purpose:

Unlocks memory locked by **ioi_physr_e**.

Parameters:

l_iocl	Type iocbT* . The virtual IOCL address.
n_iocls	Type int . Virtual IOCL size (number of IOCBs).

Return value:

0	Success.
EFAULT	Failure due to a memory fault.
EINVAL	IOCL too long or contains a transfer control block (not supported).

ioi_rsctl_e

Call format:

ioi_rsctl_e(dcbp, errprocflags)

Purpose:

Issues a reset controller and idles affected devices. NOTE: This routine is provided as a place for future extensions. It currently does nothing and returns IS_BY.

Parameters:

dcbp Type **dcbT***. Device control block pointer.
errproclags Type **int**. Error processing options.

Return value:

IS_OK Indicates that I/O was started successfully.
IS_BY Indicates that I/O failed due to a busy controller.
IS_IP Indicates that I/O failed due to a pending interrupt.
IS_BADSTATE Indicates that the device is in the wrong state.

ioi_cd_e

Call format:

ioi_cd_e(dcbp, startiocp, opcode)

Purpose:

Issues a CD (command device) instruction to a specified class E device.

Parameters:

dcbp Type **dcbT***. Device control block pointer.
startiocp Type **iocbT***. Pointer to queue of available IOCBs.
opcode Type **int**. Opcode to instruction.

Return value:

IS_OK Indicates that I/O was started successfully.

ioi_td_e

Call format:

ioi_td_e(dcbp, opcode)

Purpose:

Issues a TD instruction to a specified class E the device.

Parameters:

dcbp Type **dcbT***. Device control block pointer.
opcode Type **int**. Opcode for a TD instruction.

Return value:

Returns the device status.

ioi_wio_e

Call format:

ioi_wio_e(dcbp, docall)

Purpose:

Polls for I/O completion. NOTE: This routine currently does nothing.

Parameters:

dcbp Type **dcbT***. Device control block pointer.

docall Type **int**. The driver interrupt handler.

Return value:

IS_OK Indicates that I/O was started successfully.

ioi_ictl_e

Call format:

ioi_ictl_e(dptp)

Purpose:

Initializes an HSD device.

Parameters:

dptp Type **int**. Device parameter table pointer for the channel.

Return value:

None.

ioi_ei_e

Call format:

ioi_ei_e(level)

Purpose:

Enables interrupts for class E devices.

Parameters:

level Type **int**. The level to enable.

Return value:

None.

ioi_di_e

Call format:

ioi_di_e(level)

Purpose:

Disables interrupts for class E devices.

Parameters:

level Type **int**. The level to disable.

Return value:

None.

3.3.3 Data Structures

UTX/32 adds **Ioed_e_optypes**, a table analogous to **Ioed_optypes**, which marks legal HSD opcodes. This table is for use by HSD-based class E device drivers (see Chapter 4, "The Generic HSD Driver").

A set of **E_*** flag bits is defined. **E_VAL** is included in the entry at **Ioed_e_optypes[opcode]** if *opcode* is a valid HSD opcode.

4 The Generic HSD Driver

4.1 Overview

This chapter provides information about HSD data structures, describes the generic UTX/32 HSD device driver, and provides information useful in customizing the generic driver.

4.2 HSD Data Structures

As described in section 3.2, "Class E I/O," a TAW is a class E data structure (see Figure 3-1). Each class E device is allowed to make its own interpretation of the TAW content. As Figure 4-1 illustrates, the HSD interprets the TAW as a pointer to an IOCL. The HSD IOCL is made up of one or more I/O control blocks (IOCBs), which contain I/O instructions.

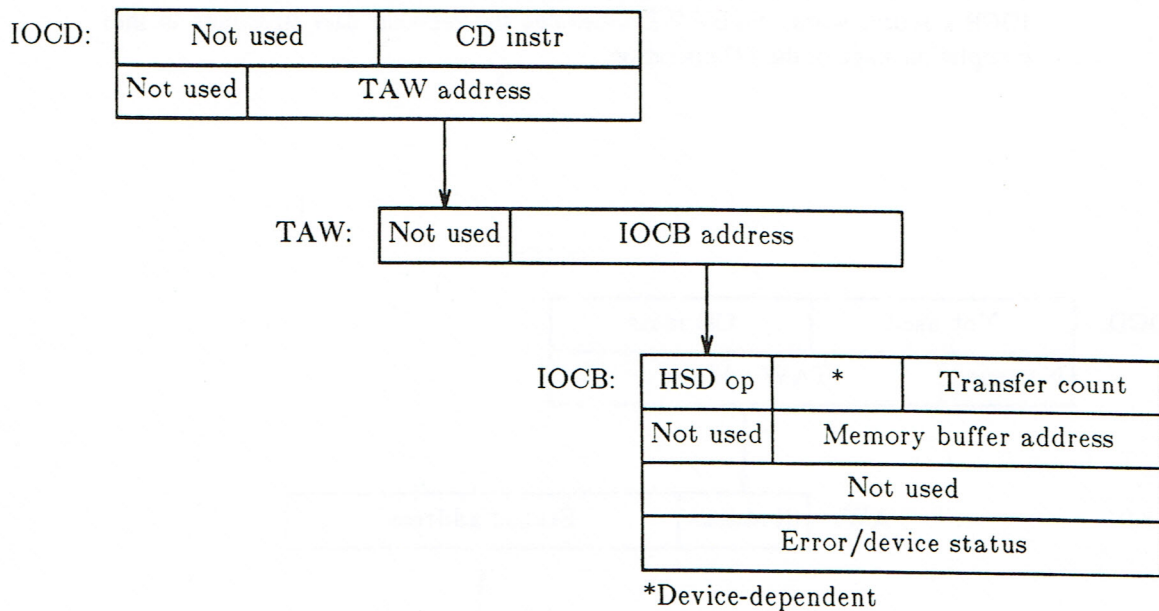


Figure 4-1. HSD Data Structures

The CD instruction includes an extended operation field specifying what is to be done with the IOCL. The operations currently supported are start I/O (SIO) and halt I/O (HIO). This field is passed to the device along with the device-dependent data field in the first word of the IOCB.

The HSD reads the IOCB and performs the operations specified by the HSD opcode in the IOCB's first word. This opcode has eight bits specifying operations to perform or not perform. There are some semantic dependencies among these bits. For details, see the *High-speed Data Interface, Model 9130/ High-speed Data Interface II, Model 9131/ High-speed Data Interface, Model 9132/ High-speed Data Inter-bus Link II, Model 9135/ High-speed Data Inter-bus Link, Model 9136 Technical Manual*.

In Figure 4-1, the second word of the IOCB is shown as a memory buffer address. This is true unless the transfer command bit is set in the HSD opcode. If the bit is set, the second word is a device-dependent data word and no other data transfer takes place.

If chaining is specified at the completion of an operation specified by an IOCB, the HSD goes on to the next IOCB in memory. If chaining is not specified, the HSD posts status and stops.

The HSD speaks directly to the memory bus using unmapped physical memory addresses. Upon completion or detection of an error, the HSD posts the service interrupt (SI) status in the TAW, and the device-completion and error status in the fourth word of the current IOCB. The SI status includes a pointer to that IOCB's fourth word. Figure 4-2 illustrates the relevant data structures in the completion stage of the I/O operation.

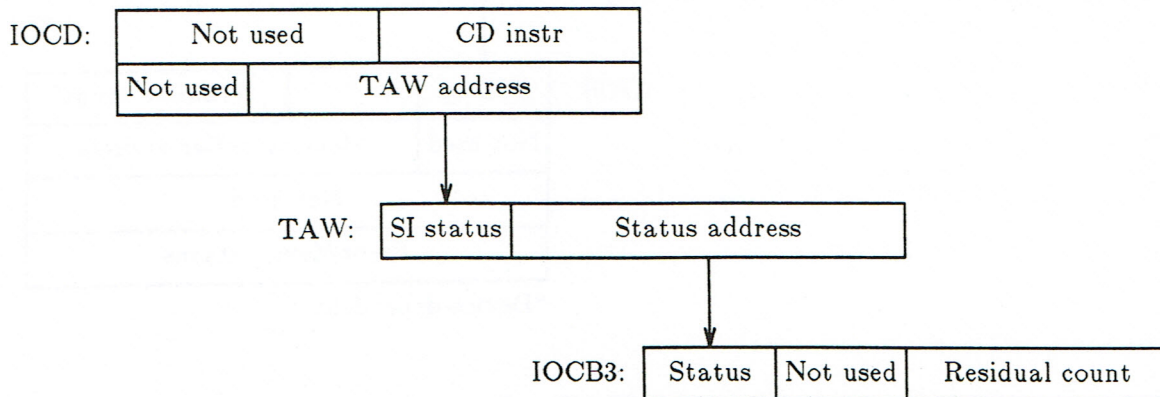


Figure 4-2. HSD Status and Error Return Structures

4.3 The Generic Driver

UTX/32 provides the generic device driver `ce` for customer devices attached to the SelBUS through the HSD. `ce` supports exclusive-use, synchronous blocking I/O using the standard driver entry points (`open`, `close`, `read`, `write`, etc.). It also supports direct I/O, described in Chapter 5. User data buffers are locked into memory during I/O (see `ce(7RT)`).

The source code in `selio/ce.c` is commented with information useful in customizing it for other class E devices. The following section discusses the main issues involved in customization.

4.4 Customization Issues

There is no standard for UTX/32 device drivers. The simplest approach to building a new device driver is to select a suitable existing driver as a model and modify that driver to suit each particular device. The UTX/32 generic HSD driver `ce` is a simple driver intended to serve as a model for custom HSD drivers. `ce` is derived from the `mpci` driver and from a special-purpose HSD driver built to support the HYPERchannel™ network interface.

This section contains hints and suggestions for building a custom driver based on `ce`. The discussion assumes that you have some knowledge of the UTX/32 kernel, or at least general UNIX® kernels.

In doing your own customization, have the following documents on hand:

- If your machine is in the PN6000 series, *Gould CONCEPT 32/67 Reference Manual*
- If your machine is in the PN9000 series, *Gould CONCEPT 32/97 Computer Basic System*
- *High-speed Data Interface, Model 9130/ High-speed Data Interface II, Model 9131/ High-speed Data Interface, Model 9132/ High-speed Data Inter-bus Link II, Model 9135/ High-speed Data Inter-bus Link, Model 9136 Technical Manual*
- *UTX/32 BSD Programmer's Reference Manual*
- The hardware technical reference manual for the device you wish to configure

4.4.1 Overview of Issues

Following are design decisions you must make for your driver:

1. Whether data is to be copied in and out of kernel buffers or buffered through user-provided space.

2. Whether the driver is to work synchronously or asynchronously (use blocking or nonblocking I/O).
3. Whether commands will use **open/close/read/write** semantics or **ioctl** semantics.
4. Whether access to the device will be shared or exclusive.

Additional design problems may stem from the special needs of your device.

The following subsections address these basic design issues and how to configure your device.

NOTE: When the driver detects a serious internal inconsistency when using kernel **printfs**, the kernel **printf** function is used to display information on the console. Kernel **printfs** interfere with normal system operation, however, and should only be used during system startup, to report serious problems, or when a panic is imminent. Kernel **printfs** should never be used in places where the call might be repeated, such as in an error condition that could be cleared but then might recur.

4.4.2 Buffering Approaches

Reading and Writing in User Space

The generic driver reads and writes data using memory space provided by the user. The data is not transferred into kernel-owned buffers before its address is supplied to the device. The user supplies the virtual address of the data. The HSD, however, can only issue unmapped memory requests: its requests must specify actual physical addresses rather than virtual addresses. The HSD support routines in the IOI build a new version of the IOCL using space provided by the driver, in which virtual addresses have been converted to physical addresses. The IOI also locks the memory pages containing the data so that the correspondence between virtual and physical addresses cannot change during the operation. When an I/O operation completes, the driver calls another IOI routine to unlock the memory pages.

This approach allows the user to provide whatever amount of memory is needed; any kernel buffering scheme will have to do reblocking of data for large requests. In designing your driver you will have to determine what requests are going to be made to your device and what the critical timing consideration is. The fastest transfers use buffers in user memory in a process that has done a **pllock** operation on itself (see *pllock(2RT/RF)*).

Using Kernel Buffers

The UTX/32 kernel provides two data structures for buffering: *bufs*, which can be allocated dynamically or statically, and *mbufs*. The possible approaches in using kernel buffers are described below. If you use kernel buffers, base your approach on the device you are interfacing, the data transfers that will be done through it, and your speed requirements.

Using bufs

The fields of a buf make it especially suited to use as a buffer. If you choose to use kernel buffering, you will want to use buf structures, whether you choose to allocate them dynamically or statically. The code for the MPCPI device driver in `selio/mpci.c` is a good introduction to the use of buf structures.

a. Dynamically allocated bufs

Disk devices typically use dynamically allocated bufs obtained from a buffer-free list and cached in a buffer cache. The disk I/O system uses a complicated buffer cache to avoid rereading recently read or written blocks. In reading the drivers for disk devices, be aware of the buffering scheme and of the separation of "strategy" routines from command routines. Disk device commands are queued and passed to lower level strategy routines that can rearrange the execution order of commands to take advantage of disk geometry. This complexity is not likely to be worthwhile in most HSD applications.

b. Statically allocated bufs

Drivers for simple devices usually use statically allocated kernel buffers. The buffers are usually buf structures allocated in the per-device data structure (for the generic driver, this is the `ce_device` structure). You may want to allocate more than one buffer to devices with large amounts of data to transfer; some drivers allocate multiple buffers in the per-device data structure, and some allocate chains of buffers using the linking fields provided in the buf structure. For example, the MPCPI driver `mpci.c` allocates two bufs per device and does double buffering, switching between buffers for consecutive operations.

If you choose to do multiple buffering, you will need to (1) keep track of the state of each buffer, (2) find a free buffer when you need one, and (3) be able to block when you run out of buffer space. For many simple devices, the gains from multiple buffering do not reward the effort. For devices with high transfer rates and low tolerance for delay, it may be absolutely critical.

Using mbufs

Network device drivers typically use mbufs because the networking code uses them, and the data would otherwise have to be rebuffered. mbufs are also appropriate for networks because they are designed to be dynamically allocated and released, to be held in queues, and to be filled and then passed to higher level routines.

The driver can call on existing routines to get and release mbufs and to manipulate their contents. For simple drivers, however, mbufs are usually unnecessarily complicated because one must worry about allocation and deallocation, and because the access routines are designed for the needs of the networking code and may be inconvenient for the needs of a simple device driver. If you have a source software license, see the Ethernet device driver code in `selio/en.c` for examples of this style.

4.4.3 Blocking and Nonblocking I/O

The generic driver uses blocking I/O semantics. Since the device itself is asynchronous, starting I/O on the device and then sleeping until the completion interrupt is returned accomplishes blocking. This means that each command is completed before the driver returns to the user. The driver assumes that only one process may issue commands for the device and that this process will not issue any additional commands while an operation is in progress. In reading the generic driver, be aware of this fundamental design decision.

You may need to support asynchronous semantics in your driver. If so, you will need to do the following:

- Decouple the issuing of commands from their execution, usually through command queueing
- Add checks and state-maintenance code to your driver
- Handle sleeps and wakeups correctly
- Provide some way of checking for I/O completions

The following subsections discuss these tasks.

Managing Command Queues

The decoupling of command-issuing from command execution is usually accomplished by queuing commands. Incoming requests are put on the queue and executed, in turn, when the device is available.

Most drivers using command queues add a **start** entry point called internally by any routine adding work to the queue or recognizing completion of an operation. The **start** routine checks that the device is idle and, if so, starts execution of the first command in the queue. In writing this routine, make sure that it can be called regardless of the state of the queue or the device.

Calling **start** when the queue is empty or when the device is busy must not cause confusion.

Since the device may complete I/O and enter its interrupt routine any time, if you are in the middle of adding a command to the queue and the device tries to execute the command, disaster may result. Therefore, block interrupts while you are modifying the command queue. This is done by first building the data block describing the operation to be put on queue, blocking interrupts, adding the block to the queue, and unblocking interrupts. Similar protection is needed when removing a block from the queue or modifying the contents of a block on the queue.

Using a State Machine Model

If your driver is to be asynchronous, you will probably want to build it as a *state machine*. A state machine driver has a single variable recording the state of the device. Each event affecting the device alters the value of the variable. At each device event (interrupt, command-queue change, or time out), you can use the device state and the event to determine whether the event is legal and what the new state should be.

A *state transition diagram* is a convenient means of reviewing whether your driver will work as intended. The diagram represents the states as nodes in a graph, with arcs representing legal transitions. The arcs are labeled with the event causing the transition, in this case usually a function call name.

A simplified state transition diagram for the generic driver is shown in Figure 4-3.

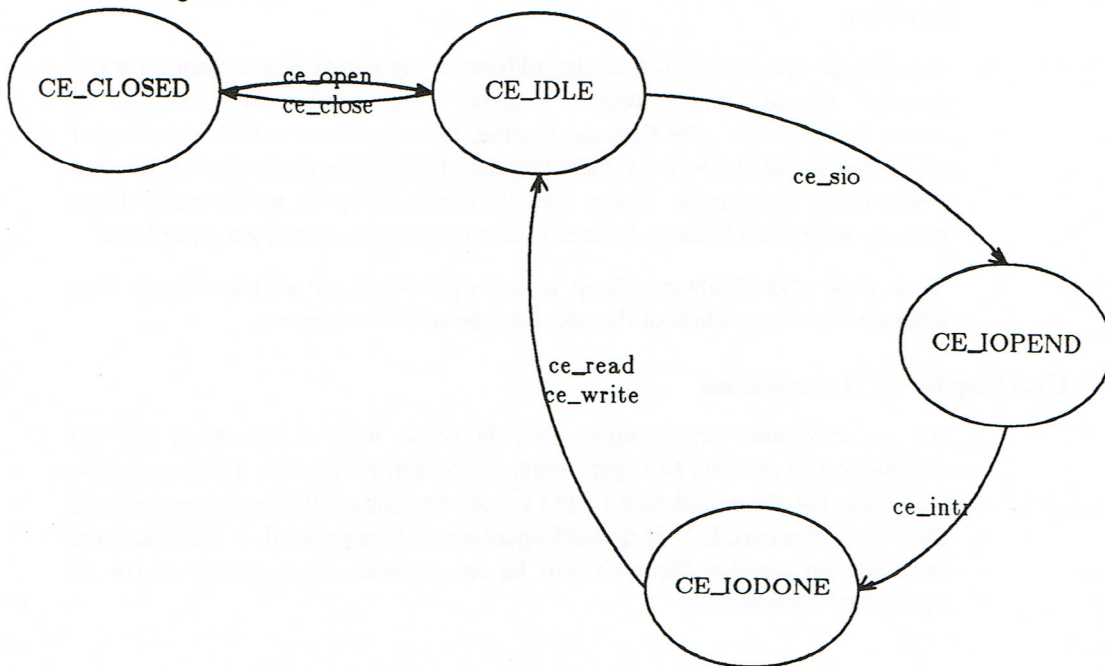


Figure 4-3. State Transition Diagram for the Generic HSD Driver

At each entry point, do two things: check that the device is in a reasonable state for the requested operation, and change the state appropriately. If you do command queuing, you can concentrate the state checks in the start and interrupt routines, since a requested operation can be queued without checking state.

Care in constructing the state transition scheme and care in coding so that the transition takes place under well-defined conditions and in orderly places will be rewarded.

The advice given in the previous section, "Managing Command Queues," about blocking interrupts around queue changes also applies to state changes. You must not allow another process, which could be the interrupt handler for your device, to run while your state is inconsistent or misleading.

Handling Sleeps and Wakeups

To provide asynchrony, you need to be aware of the way **sleep** and **wakeup** are used in UTX/32. The **sleep** call puts a process into a list of sleeping processes and provides a name under which the process will sleep. The **wakeup** call wakens all processes sleeping under a particular name.

In building your asynchronous driver, you will need to select a convention for naming sleeping processes. Like many other UTX/32 drivers, the generic driver uses a single name for all its sleeps and wakeups. That name is the address of the beginning of the table of per-device data structures for devices using the driver. This naming convention simplifies the data-passing requirements, since that address is known everywhere **sleep** or **wakeup** may be called.

A more specific name, such as the address of the per-device structure for the specific device, might require making more information visible at inconvenient times. The **timeout** routine, for instance, is called with limited information and might have difficulty identifying the particular device and constructing its address. A less specific name, however, would mean that a wakeup might reach many devices whose operations are not yet completed.

Your code should always sleep in a loop testing for a state change that indicates the completion of the specific operation in progress.

Checking for I/O Completions

An asynchronous driver must provide some way of checking for the completion of pending I/O operations. Some drivers provide a **wio** operation that waits for the completion of I/O before returning, allowing users to build their own blocking I/O. If queued operations are supported, a decision must be made on whether the wait will be for a particular operation or for all operations (quiescence).

Some drivers allow the user to specify whether a command is to block or not. To provide that option, build the driver to support nonblocking I/O, then check a flag in the user's request and sleep if so requested. The sleep loop will have to be more complicated than described in the previous section, "Handling Sleeps and Wakeups," since it will have to check for completion of the specific operation requested.

4.4.4 Command Semantics

The generic driver supports standard UTX/32 **open**, **close**, **read**, **write**, and **ioctl** semantics. This provides a reasonable model of the use of many devices, but it may be inappropriate for others. The following subsections discuss the more extensive use of **ioctl** semantics and special considerations for networking.

ioctl Semantics

If your driver really controls a device rather than doing I/O to it, you might want to use purely **ioctl** semantics. If you do, your driver must contain dummy routines for the unused routines and a more complicated **ioctl** routine.

A device is a good candidate for this model if its operations are, for the most part, insensitive to previous operations. If your device is, for instance, a one-way device such as a sensor or actuator, the **ioctl** interface might be the simplest basis for the driver. The device-status request **ioctl** in the generic driver is an example of how to build commands in such a driver.

Network Device Semantics

Network devices generally have a semantics based on a queuing model, since packets are passed to and from the driver on queues. A network device must be able to accept data that was not requested by a user, and to dispense data when it is requested. It must also be able to manage communications with the other end of the link so that when queues are full, communications can be suspended.

Instead of being driven by user commands like **read** and **write**, a network driver is usually driven on one side by the availability of data from the network, and on the other side from the upper protocol layers. The driver and protocols provide entry points for getting data on and off queues and for informing the other side that the queue has been modified.

In some cases such as with the Ethernet **en** driver, the driver also includes **read** and **write** entry points that can be used for direct access to the device. The semantics of such direct access may be confusing. For example, the **en** driver **read** routine delivers an incoming packet to all users with outstanding reads on the device.

Most network drivers contain more code supporting networking than supporting the device. If you are going to write a driver for a network device, use an existing network device driver as a starting point, and use the low level routines in `selio/ce.c` to provide services to the higher level routines in the network device driver. The Ethernet `en.c` and SCM `scm_if.c`, etc. network devices are possible models.

4.4.5 Shared and Exclusive Device Access

The generic driver is explicitly an exclusive-use driver. This means that

- Only one process can use it at a time.
- Before being used, it must be opened by the user process.
- When no longer needed, it must be closed by the user process.

These assumptions simplify things for the driver, since it can determine things about the user process when the device is opened and not do so again.

Many devices, however, are not or need not be used exclusively, or are used for one command rather than for a series of commands. If your device will be used on such basis, you will have to do more user validation in individual command routines.

If you will not be using `open` and `close` to define user sessions, replace these routines with dummies that return `NULL`. If, on the other hand, you allow multiple users but still want to require each user to specifically bracket its use of the device, use the `open` routine to build a block defining a session, validate in all other operations that a valid session exists for the calling process, and use the `close` routine to delete the session definition.

If shared use is permitted, the driver must take ownership into account in sequencing operations. Only the process initiating an operation, for instance, should be allowed to wait for the completion of that operation.

If you are writing a shared-use device driver, take all the precautions described for manipulating command queues (see "Managing Command Queues") or the device state (see "Using a State Machine Model") in the per-device data structure. Assume that one process may be suspended and another may run any time you have not specifically blocked interrupts. A partially completed queue modification or state change could cause disaster.

Network devices are, again, an exception. They usually provide a transport service used by higher level protocols. They have no notion of users. It is assumed that the protocol layers will assign data to specific users and that the device will pass along whatever identifying address information the protocols provide as part of the data.

4.4.6 Configuring the Device

To make your device available to UTX/32 users, do the following:

1. Add the device to the CONFIGURATION file for your system so that the necessary kernel table entries are built.
2. Add your driver to the makefile in the **selio** directory.
3. Make the device entries for your device(s).

UTX/32 gets to the routines in your device driver through several kernel data structures, including **cdevsw**, interrupt service routines, the device parameter table **Dpt**, and the mounted device **mtd** tables. The **config** program (see *config(8)*) builds the necessary table entries based on entries you insert in the configuration file it reads.

To provide flexibility in assigning physical devices to device drivers, **config** requires you to provide the driver name in the CONFIGURATION file entry. The entry

```
hsd ce0 at 0x4000 priority 0x12
```

tells **config** that there is an HSD device at 0x4000 with priority 0x12 and that it will be minor device 0 on the device driver named **ce**.

If you are interfacing multiple devices of the same type which will be used through the same driver, give them successive minor device numbers with the same driver-name prefix. Names of devices to be accessed through a different driver would have a different prefix.

The system-building process must be aware of your driver. Add it to the list of source files in the makefile in the **selio** directory (the list named SRCS) and to the list of sources in the master makefile in the /sys directory (the list named SELIOSRC). Do a **make depend** in the **object** directory after changing the makefiles.

The **config** program will not automatically add your device to **madev.sh** because the script does not have enough information to determine your device's address. Use the major and minor device numbers reported in the configuration report to run **mknod** for your devices. Since this must be done whenever your system is built, add these **mknod** lines to **madev.sh** so that they will be run automatically.

5 Direct Input/Output

5.1 Overview

The direct I/O facility (DIO) provides enhanced I/O services to real-time UTX/32 processes.

In a typical UTX/32 system, the same system calls (**open**, **close**, **read**, **write**, etc.) are used for all I/O, whether the interaction is with a file, terminal, tape, or communications pipe. This uniform handling of I/O, while appropriate in a time-sharing environment, is not suitable in a real-time environment for two reasons. First, the operating overhead for these calls is significant, including traversing multiple tables, sharing devices among processes, buffering data in the kernel, translating between virtual and physical addresses, and constructing IOCLs. Further, the response to any given I/O request is unpredictable, since all requests are subject to standard time-sharing prioritization and swapping rules.

UTX/32 avoids these problems by giving real-time processes direct access to I/O devices. The gain is a higher I/O bandwidth at reduced CPU overhead.

DIO provides real-time processes with the following features:

- The ability to issue I/O commands directly to devices
- I/O service according to real-time priority
- Several mechanisms for notification of I/O completions
- The ability to connect to and disconnect from devices
- The ability to reserve devices

This chapter discusses each of these features.

The DIO routines mentioned here are described in detail in Chapter 3 of the *UTX/32 BSD Programmer's Reference Manual*. DIO is based on the DIO device driver (see *dio(7RT)*).

WARNING: Careless use of DIO facilities may result in damage to data or failure of the system.

5.2 Application

DIO is usable with any class E or class F device that has a UTX/32 device driver with a functional maintenance interface entry point. The only requirement is that the real-time process must be locked into memory (see *plock(2RT/RF)*).

5.3 Issuing I/O Commands

A process requesting I/O must construct an IOCL, request that the IOCL be executed, and perform the appropriate error-recovery.

On Gould PowerNode™ hardware, IOCLs must meet two requirements: they must be constructed with physical memory addresses, and be physically contiguous. Since UTX/32 hides knowledge of page boundaries and virtual-to-physical address mapping, DIO provides real-time processes with IOCL translation services for both requirements. Note that these translations can increase the size of the IOCL.

5.3.1 Virtual to Physical Address Translation

Virtual addresses can be translated into physical addresses. The process can proceed in either of two ways when building the IOCL:

- Build the IOCL with virtual addresses, convert these addresses into physical addresses using **dioconvert** (see *dioconvert(3RT/RF)*), and then issue the IOCL using **diosiophys** (see *diosio(3RT/RF)*).
- Use virtual addresses and issue the IOCL using **diosiolog** (see *diosio(3RT/RF)*). DIO will do the translation.

In either case, the I/O is performed directly out of and into the user's address space; it is not buffered by or copied into the kernel. The I/O completion status is returned into a status buffer specified by the user. The user must check that the I/O completed successfully and take appropriate error-recovery measures.

5.3.2 Ensuring Physical Contiguity

The user must ensure physical contiguity by placing an IOCL entirely on a page. DIO uses data chaining to accommodate I/O requests specifying user data extents that cross page boundaries.

5.3.3 Disabling Kernel Checking

To enhance performance, the user can disable the kernel's checking of the validity of IOCLs (see *diosio(3RT/RF)*).

WARNING: Use this option with extreme caution. IOCLs contain physical memory addresses. Therefore, an incorrect IOCL can easily destroy the integrity of other processes or the kernel. The destruction may be virtually impossible to associate with an incorrect IOCL. Further, if IOCL checking is disabled, the check for physical contiguity is also disabled.

If the user decides to disable kernel checking, it should be done only after the IOCL-manipulation code has been thoroughly tested.

5.4 Priority Ordering

On Gould PowerNode hardware, a device subchannel can have only one active request at any given time. The driver queues any additional I/O requests according to the real-time priority of the requesting process. Within each priority, I/O requests are serviced in FIFO order. When an I/O completes, the next queued IOCL is issued. A connection request can be aborted with **dioabort** (see *dioabort(3RT/RF)*).

5.5 I/O Request Tracking

Real-time processes using DIO can track their I/O requests and govern whether and how to be notified of I/O completions.

5.5.1 Identifying Outstanding Requests

Real-time processes can track outstanding I/O requests using the two counters maintained for each connection. **io_initiated** is the number of I/O requests accepted by DIO. **io_completed** is the number of I/O requests that have been serviced successfully, with I/O errors, or cancelled. The difference between the counters is the number of I/O requests outstanding. (Signed arithmetic works correctly even when the counters wrap.)

Since most devices allow only one active I/O request per subchannel, and I/O requests on a busy device are queued in FIFO order, a process is aware of pending requests. For devices that can handle multiple requests, a status buffer associated with each request can be examined to determine the completed I/Os.

5.5.2 Notification of I/O Completions

The following I/O completion notification mechanisms are settable using **dionotify** (see *dionotify(3RT/RF)*):

Wait I/O

The calling process is blocked until the I/O completes.

No-wait I/O, no completion notification

The calling process continues after issuing the I/O request. The process does not receive any asynchronous notification. However, the process can poll the above-mentioned two counters to determine when the I/O completes.

No-wait I/O, completion notification via signals

The calling process continues after issuing the I/O request. The process receives a signal when one or more I/O requests complete. Since the completion of one or more I/O requests can be reported with a single signal, the process should check the two counters to determine the number of I/O completions.

A threshold value specifies when signals are sent. When an I/O operation completes and the number of outstanding I/O requests is at or below the threshold, a signal is generated. For example, a value of zero indicates the user wants a signal only when all outstanding I/O requests have completed. A large value such as 100 indicates the user wants a signal each time an I/O completes.

5.6 Connecting and Disconnecting

A real-time process **pl**ocked into memory can connect to a specified device and subchannel by issuing the connect command **dioconnect** (see *dioconnect(3RT/RF)*). If the device is not being used by UTX/32 for another purpose, UTX/32 relinquishes it to the real-time process.

Once established, the connection remains in effect until the real-time process issues the disconnect command **diodisconnect** (see *diodisconnect(3RT/RF)*). All pending I/O operations on a connection are flushed when disconnection occurs.

5.7 Reserving a Device

Typically, UTX/32 has access to a device between DIO connections. Sometimes, however, a real-time process requires greater predictability than this arrangement allows — it needs guaranteed access. DIO therefore allows for a device to be reserved for the exclusive use of a single process. If a device is reserved using **dioreserve**, DIO acquires the specified device from UTX/32 when the first connection to the device is established, and returns it to UTX/32 only when the device is released with **diorelease** and the last connection is closed (see *dioreserve(3RT/RF)* and *diorelease(3RT/RF)*).

Appendix A

Input/Output Interface Specification Files

The following sections contain specification files for the IOI.

A.1 Sample Device Driver Interface

The appropriate tables, table entries, and interrupt routines (CDEVSW, BDEVSW, ICB, DPT, SRT, DCBs, and ISR) must first be built during the system generation process.

The device driver must provide the following entry points:

<i>dev_open</i>	Block or character device
<i>dev_close</i>	Block or character device
<i>dev_init</i>	Block or character device
<i>dev_read</i>	Block or character device
<i>dev_write</i>	Block or character device
<i>dev_strategy</i>	Block device
<i>dev_ioctl</i>	Character device
<i>dev_intr</i>	Block or character device
<i>dev_maint</i>	Block or character device

The code in the examples below is not mandatory, but is intended as a guide to the use of the IOI.

dev_init

The driver may choose to initialize the private use fields of its DCBs at this time.

dev_open

The driver obtains a pointer to DCB by indexing *dev_mtd*[]. It obtains a pointer to its own private control block for that device by accessing the private use field of the DCB.

dev_close

The driver performs cleanup. This may include calling *ioi_halt*().

dev_read

The driver performs any internal bookkeeping necessary, then calls *physio*() which in turn calls the driver's strategy routine.

dev_write

The driver performs any internal bookkeeping necessary, then calls *physio*() which in turn calls the driver's strategy routine.

dev_strategy

The driver places the request on its private queue. If the device is currently inactive, the driver calls its internal start routine, which pulls the request off the queue and calls `ioi_sio()`. In the case of new I/O, `ioi_phys()` is called either from the strategy routine or from the start routine. Note, however, that `ioi_phys()` requires a process pointer for the process requesting I/O, and that the start routine may be called on the interrupt stack.

dev_ioctl

The driver updates the internal state appropriately and perhaps does I/O.

dev_intr

The driver parses result code found in DCB. If no error is present, a wakeup is performed on previous I/O, and the next I/O is started by calling the driver's internal start routine. The start routine pulls the next entry off the driver's queue and calls `ioi_sio`.

If there is an error, the sense buffer status is examined to determine if sense information was available.

If no sense information was obtained, the device is declared broken, the IOCL is deallocated by calling `ioi_dalloc()`, and the interrupt routine exits.

If sense information is available, the sense buffer is examined to determine the reason for the I/O failing. Device-dependent action is then performed.

dev_maint

If the reason parameter indicates a REQUEST, the device driver determines if the specified subchannel can be relinquished at this time.

If it can be relinquished, the device driver notes internally that the device is in use by a maintenance process and returns zero or EBUSY.

If the reason parameter indicates RELEASE, the device driver marks the subchannel available. All subchannels associated with a physical device must be released before I/O to that physical device is performed.

A.2 sel/selio.h

```

#ifndef Hselio
#define Hselio 1

/*****
 *          CLASS F I/O COMMAND DOUBLEWORD (iocd) *
 *****/
typedef struct iocd iocdT;

struct iocd
{
    unsigned char   ioc_cmd;           /* COMMAND           ( 8 BITS) */
    int             ioc_addr:24;       /* PHYSICAL ADDRESS  (24 BITS) */
    unsigned char   ioc_flg;          /* FLAG BITS        ( 8 BITS) */
    unsigned char   ioc_junk;         /* A HOLE (MUST BE ZERO) ( 8 BITS) */
    unsigned short  ioc_cnt;          /* BYTE COUNT       (16 BITS) */
};

/*
 * CLASS F IOCD FLAG FIELD DEFINES
 */
#define IOC_DCHAIN      0x80          /* DATA CHAIN      */
#define IOC_CHAIN      0x40          /* COMMAND CHAIN    */
#define IOC_SIC        0x20          /* SUPPRESS INCORRECT LENGTH */
#define IOC_SKIP       0x10          /* SKIP             */
#define IOC_PCI        0x08          /* PROGRAM CONTROLLED INTERRUPT */
#define IOC_RTI        0x04          /* REAL TIME OPTION */
#define IOC_NU1        0x02          /* NOT USED -- MUST BE ZERO */
#define IOC_NU2        0x01          /* NOT USED -- MUST BE ZERO */

#define IS_CHAINED(x) ( (x) & (IOC_DCHAIN | IOC_CHAIN) )

/*
 * CLASS F IOCD OPCODE FIELD DEFINES
 * (Generic and non-generic opcodes)
 */
#define OP_INCH        0x00          /* ALL DEVICES - INITIALIZE CHANNEL */
#define OP_WRIT        0x01          /* ALL DEVICES - WRITE DATA */
#define OP_READ        0x02          /* ALL DEVICES - READ DATA */
#define OP_NOP         0x03          /* ALL DEVICES - NO OPERATION */
#define OP_SENS        0x04          /* ALL DEVICES - SENSE */
#define OP_SEEK        0x07          /* UDP DISC - SEEK (CYL, TRK, SEC) */
#define OP_LINEC       0x07          /* IOP SCM - LINE CONTROL */
#define OP_TIC         0x08          /* ALL DEVICES - TRANSFER IN CHANNEL */
#define OP_RDECHO      0x0a;        /* CONSOLE - READ WITH ECHO */
#define OP_WAIT        0x0b          /* IOP SCM - WAIT (NOT IMPLEMENTED) */
#define OP_CONN        0x0f          /* IOP SCM - CONNECT */
#define OP_LPL         0x13          /* UDP DISC - LOCK PROTECT LABEL */
#define OP_SENTC       0x14          /* IOP SCM - SENSE TRANSFER COUNT */
#define OP_DISC        0x1f          /* IOP SCM - DISCONNECT */
#define OP_LMR         0x1f          /* UDP DISC - LOAD MODE REGISTER */
#define OP_RES         0x23          /* UDP DISC - RESERVE DRIVE */
#define OP_OBTN        0x24          /* IOP SCM - OBTAIN LINK STATISTICS */

```

```

#define OP_IDENT          0x2f
#define OP_REL            0x33 /* UDP DISC - RELEASE DRIVE */
#define OP_XEZ            0x37 /* UDP DISC - RECALIBRATE HEADS */
#define OP_RSET           0x4f /* IOP SCM - RESET LINK STATISTICS */
#define OP_SRM            0x4f /* UDP DISC - SET RESERVE TRACK MODE */
#define OP_RTL            0x52 /* UDP DISC - READ TRACK LABEL */
#define OP_XRM            0x5f /* UDP DISC - RESET RESERVE TRACK MODE */
#define OP_SETP           0x7f /* IOP SCM - SET LINK PARAMETERS */
#define OP_ECC            0xb2 /* UDP DISC - READ ECC INFORMATION */
#define OP_LACS           0xf1 /* IOP SCM - LOAD ACS */
#define OP_RACS           0xf2 /* IOP SCM - READ ACS */
#define OP_ICH            0xff /* IOP/UDP/TAPE- INCH */
#define OP_MODE           0xff /* IOP SCM - MODE CONTROL */

/*
 * CLASS-F IOCD COMMAND FIELD DEFINES FOR IPI HOST ADAPTER
 */
#define HA_INCH           0x00 /* ALL DEVICES - INITIALIZE CHANNEL */
#define HA_WD             0x01 /* ALL DEVICES - WRITE DATA */
#define HA_RD             0x02 /* ALL DEVICES - READ DATA */
#define HA_NOP            0x03 /* ALL DEVICES - NO OPERATION */
#define HA_TIC           0x08 /* ALL DEVICES - TRANSFER IN CHANNEL */
#define HA_RDB            0x0c /* ALL DEVICES - READ DATA BACKWARDS */
#define HA_ASB            0x20 /* ALL DEVICES - ALLOCATE STATUS BUFFER */
#define HA_WRAP           0x30 /* ALL DEVICES - DIAGNOSTIC WRAP TEST */
#define HA_HABU           0x40 /* ALL DEVICES - HOST ATTRIBUTE UPDATE */
#define HA_FIFW           0xb0 /* ALL DEVICES - DIAG FIFO WRITE TEST */
#define HA_TSP            0xe3 /* ALL DEVICES - TRANSFER SLAVE PACKET */
#define HA_FIFR           0xf0 /* ALL DEVICES - DIAGNOSTIC FIFO READ TEST */
#define HA_TCP            0xf3 /* ALL DEVICES - TRANSFER COMMAND PACKET */

/*
 * IPI FLAG FIELDS FOR RESET CONTROLLER INSTRUCTION
 * (PLANTED IN THE IOCLA FIELD OF THE DEVICE'S ICB)
 * MUST BE ISSUED TO BRING IPI SLAVES OUT OF MAINTENANCE MODE
 */
#define IPI_RSCTL_LI      0x02 /* RESET LOGICAL INTERFACE */
#define IPI_RSCTL_PI      0x04 /* RESET PHYSICAL INTERFACE */

#define IPI_RSCTL_SOFT ( (iocdT *)IPI_RSCTL_LI )
#define IPI_RSCTL_HARD ( (iocdT *) (IPI_RSCTL_LI | IPI_RSCTL_PI) )

/*
 * A SIMPLE MACRO TO DUMP I/O COMMAND LISTS (IOCLs)
 * (note: the macro argument is modified)
 */
#define DUMP_IOCL(sp) \
do \
printf("\t(%x) %x %x\n", (sp), *(int *) (sp), (((int *) (sp))+1)); \
while ( IS_CHAINED((sp)++->ioc_flg))

```



```

/*****
 *      CLASS F I/O STATUS DOUBLEWORD (iostatus)      *
 *****/
typedef struct iostatus iostatusT;

struct iostatus
{
    unsigned char   ios_subaddr;           /* DEVICE SUBADDRESS COMPLETING */
    int             ios_iocdp:24;         /* TERMINATION IOCD (+1 IOCD) */
    short           ios_flags;           /* CHANNEL/DEVICE STATUS FLAGS */
    unsigned short  ios_resbytec;        /* RESIDUAL BYTE COUNT */
};

/*
 * I/O STATUS DOUBLEWORD FLAG FIELD DEFINES
 */
#define IF_ECHO           0x8000          /* ECHO */
#define IF_PCI           0x4000          /* PROGRAM CONTROLLED INTERRUPT */
#define IF_INCOR_LENGTH  0x2000          /* INCORRECT LENGTH */
#define IF_CPCHECK       0x1000          /* CHANNEL PROGRAM CHECK */
#define IF_CDCHECK       0x0800          /* CHANNEL DATA CHECK */
#define IF_CCCHECK       0x0400          /* CHANNEL CONTROL CHECK */
#define IF_IFCHECK       0x0200          /* INTERFACE CHECK */
#define IF_CHCHECK       0x0100          /* CHAINING CHECK */
#define IF_BUSY          0x0080          /* BUSY */
#define IF_STATMOD       0x0040          /* STATUS MODIFIER */
#define IF_CTLEND        0x0020          /* CONTROLLER END (NOT USED) */
#define IF_ATTEN         0x0010          /* ATTENTION */
#define IF_CE            0x0008          /* CHANNEL END */
#define IF_DE            0x0004          /* DEVICE END */
#define IF_UC            0x0002          /* UNIT CHECK */
#define IF_UE            0x0001          /* UNIT EXCEPTION */

#endif

```

A.3 selio/loi.h

```
/*
 *      (c) Copyright 1986 Gould Inc.
 *      All Rights Reserved.
 */

/*      @(#) UTX/32 2.0  loi.h ver 2.0 */

/*
 *      @(#) -- file loi.h, version 2.0.  Last changed 5/22/86 18:25:08.
 */
#ifndef Hloi
#define Hloi      "@(#)loi.h      2.0"

#ifndef Hselio
#ifdef KERNEL
#include "../sel/selio.h"
#else NOT KERNEL
#include <sel/selio.h>
#endif KERNEL
#endif Hselio

#ifndef Hsystypes
#ifdef KERNEL
#include "../h/types.h"
#else NOT KERNEL
#include <sys/types.h>
#endif KERNEL
#endif Hsystypes

#ifdef KERNEL
#include "../sel/psd.h"
#include "../sel/icb.h"
#else NOT KERNEL
#include <sel/psd.h>
#include <sel/icb.h>
#endif KERNEL
#endif Hloi
```



```

/*****
*          SOME USEFUL DEFINES
*****
#define spl4      spl5          /* ACI / DACI ON IOP DOESN'T WORK RIGHT */
#define ROUNDUP(thing,boundary) ((thing + (boundary-1)) & ~ (boundary - 1))
#define STATE(p) (p)->d_ioistate
#define ROUNDUP(thing,boundary) ((thing + (boundary-1)) & ~ (boundary - 1))
#define CN_DPTINX      1          /* GUARANTEED BY config(8) */

#ifdef SYS5
#define Return(x) {u.u_error = x;return;}
#else SYS5
#define Return(x) return (x);
#endif SYS5

#define RETRYCNT      4          /* NUMBER OF TIMES TO RETRY I/O OPERATIONS */
#define RSCHNLNT      300        /* DELAYS FOR RESET-CHANNEL TO COMPLETE */
#define INCHRETRYCNT  16         /* NUMBER OF TIMES TO RETRY INCH OPERATION */
#define SNS_TIME      5          /* SECONDS TO WAIT FOR SENSE TO COMPLETE */
#define TIME_OUT      60         /* # OF CLOCK TICKS BETWEEN TIME OUTS */

#define DELAY_SHORT    2000       /* SHORT BUZZ LOOP DELAY VALUE (FOR POLL) */
#define DELAY_LONG    32000      /* LONG BUZZ LOOP DELAY VALUE (FOR INCH) */

#define IPI_INCH_T1   0          /* IPI SLAVE SELECT WAIT TIME (0=DEFAULT) */
#define IPI_INCH_T2   0          /* IPI REQ INTERR WAIT TIME (0=DEFAULT) */

#define WIO_DCALL     0          /* CALL INTERRUPT ROUTINE FROM ioi_wio() */
#define WIO_DONTCALL  1          /* DON'T CALL INTERRUPT ROUTINE... */

#define DCB_NULL 0          /* NULL ENTRY */

/*
*   HARDWARE DEVICE ADDRESS MASKS
*/
#define M_CHANNEL      (0xFF00)   /* SelBUS CHANNEL FIELD MASK */
#define M_CONTROLLER  (0x00F0)   /* SelBUS CONTROLLER FIELD MASK */
#define M_DEVICE       (0x000F)   /* SelBUS DEVICE FIELD MASK */
#define M_SUBCHANNEL   (M_CONTROLLER | M_DEVICE)

/*
*   HARDWARE DEVICE ADDRESSING
*/
#define CHANNEL(c)     (c & M_CHANNEL) /* SelBUS CHANNEL ADDR */
#define CTRLR(c)       (c & M_CONTROLLER) /* SelBUS CONTROLLER ADDR (MUXs) */
#define DEVICE(c)      (c & M_DEVICE) /* SelBUS DEVICE ADDR */
#define SUBCHANNEL(c) (c & M_SUBCHANNEL) /* SelBUS SUBCHANNEL ADDR */

/*
*   ICB STATUS POINTER MANIPULATION
*/
#define SET_CC_STORED(p) ((iostatp *)(((int)p) | 0x40000000))
#define CLR_CC_STORED(p) ((iostatp *)(((int)p) & 0x00ffffff))
#define CC_NOT_STORED(iostatp) (((int) iostatp) & 0xf0000000) != 0x40000000)
#define CC_STORED(iostatp) (((int) iostatp) & 0xf0000000) == 0x40000000)

```

```

/*
 *   FOR DPIO AND UDP DISC CONTROLLERS:
 */
#define DISK_BUF_SIZE (224 * 4)

/*****
 *           IOI STATE TABLE
 *****/
typedef struct iostate ioi_stateT;

struct iostate
{
    unsigned char    i_state;        /* MAJOR STATE FOR THIS SUBCHANNEL */
    unsigned char    i_queued;       /* QUEUED I/O STATUS PRESENT */
    unsigned char    i_errflags;     /* ERROR PROCESSING FLAGS */
    unsigned char    i_serrflags;    /* SPONTANEOUS INTR ERROR PROC FLAGS */
    short int        i_errcnt;       /* ERROR RETRY COUNT */
    short int        i_timeout;      /* HOW LONG BEFORE TIME OUT */
    iostatusT        i_qiostat;      /* QUEUED I/O STATUS */
    iocdT            i_snsiocd;      /* SENSE IOCD */
    unsigned char    i_maint;        /* SET TO INDICATE DCB IN MAINT. MODE */
    unsigned char    i_mreason;      /* MAINTENANCE REASON */
    unsigned char    i_mstate;       /* MAINTENANCE STATE */
    dev_t            i_dev;          /* /dev/ioi<nn> DEVICE IN MAINT. MODE */
};

/*
 * THE FOLLOWING DEFINE THE VALUES OF i_state
 */
#define IS_IDLE      0              /* IDLE */
#define IS_BUSY      1              /* I/O IN PROGRESS */
#define IS_HALTWAIT  2              /* WAITING FOR INTR AFTER HALT I/O */
#define IS_STATWAIT  3              /* WAITING FOR INTR AFTER SENSE FOR ERR */
#define IS_SPSTATWAIT 4             /* WAITING FOR INTR AFTER SENSE FOR ERR */
/* ...RECOVERY FOR SPONTANEOUS INTR */
#define IS_IGNOREINTR 5            /* BUSY ON SUBCHANNEL. IGNORE NEXT */
/* ...TO CLEAR PENDING INTR CONDITION */

/*
 * THE FOLLOWING DEFINE THE VALUES OF i_mstate
 */
#define IM_IDLE      0              /* MAINTENANCE DEVICE IS IDLE */
#define IM_BUSY      1              /* MAINTENANCE DEVICE IS DOING I/O */
#define IM_URINTR    2              /* MAINTENANCE DEVICE GOT AN INTERRUPT */

/*
 * DEFINES FOR i_errflags. ALSO USED WHEN CALLING I/O ROUTINES LIKE ioi_sio()
 */
#define IE_SUC        0x01         /* OBTAIN SENSE INFO ON UNIT CHECK */
#define IE_SUE        0x02         /* OBTAIN SENSE INFO ON UNIT EXCEPTION */
#define IE_SAT        0x04         /* OBTAIN SENSE INFO ON ATTENTION */
#define IE_SIL        0x08         /* OBTAIN SENSE INFO ON INCORRECT LENGTH */

#define IE_RTYCBY     0x10         /* RETRY IF REFUSED DUE TO BUSY DEVICE */
#define IE_HTO        0x20         /* ISSUE HALT I/O ON TIME OUT */
#define IE_RTO        0x40         /* ISSUE RSCTL ON HALT I/O TIME OUT */
#define IE_MAINT      0x80         /* MAINT MODE -- FOR IOI USE ONLY */

```



```

/*****
 *      DEVICE CONTROL BLOCK (dcb) STRUCTURE
 *****/
typedef struct dcb dcbT;

struct dcb
{
    short int    d_channel;          /* CHANNEL/SUBCHANNEL ADDRESS */
    dev_t        d_dev;              /* MAJOR/MINOR DEVICE NUMBER */
    short int    d_dptinx;           /* DEVICE PARAMETER TABLE INDEX */
    short int    d_snsct;            /* SENSE (IPI ASB) COUNT */
    caddr_t      d_sensep;           /* SENSE (IPI ASB) BUFFER ADDR */
    ioi_stateT   d_ioistate;         /* IOI STATE VECTOR */
    iocdT        *d_ioclp;           /* CURRENT IOCL POINTER */
    iostatust    d_iostat;           /* CHANNEL COMPLETION STATUS */
    int          d_priv;             /* PRIVATE USE FOR DEVICE DRIVER */
};

/*****
 *      DEVICE PARAMETER TABLE (dpt) STRUCTURE
 *****/
typedef struct dpt dptT;

struct dpt
{
    char         *dp_name;           /* ASCII NAME OF CONTROLLER */
    iocbT        *dp_icbp;           /* INTERRUPT CONTEXT BLOCK (ICB) POINTER */
    int          (*dp_intr)();        /* INTERRUPT SERVICE ROUTINE ADDRESS */
    int          (*dp_init)();        /* DRIVER INITIALIZATION ROUTINE ADDRESS */
    int          (*dp_maint)();       /* MAINTENANCE ROUTINE ADDRESS */
    int          dp_snslen;           /* SENSE (IPI STATUS BUFFER) LENGTH */
    dcbT        **dp_srtp;           /* SUBCHANNEL ROUTING TABLE POINTER */
    int          dp_nsubchan;         /* NUMBER OF SUBCHANNELS */
    short        dp_ctl_type;         /* CONTROLLER TYPE */
    caddr_t      dp_dev_buf;         /* CONTROLLER STATE (INCH) BUFFER ADDRESS */
    short        dp_chanaddr;        /* CHANNEL ADDRESS */
    short        dp_mtdlen;          /* LENGTH OF MTD (IF NON-ZERO) */
    dcbT        **dp_mtdp;          /* POINTER TO ASSOCIATED MTD */
};

/*
 * DEFINES ON dp_ctl_type (CONTROLLER TYPE FIELD)
 *
 * NOTE: CT_SELFCUSTOM is used as a 'fence' for certain tests in the IOI. It
 * must be the highest numbered controller type that is still a SelBUS
 * device. All SelBUS devices must fall below type CT_SELFCUSTOM below.
 *
 * NOTE: CT_SEL/CT_SELE and CT_IPIS/CT_IPIE are the lower/upper bounds
 * for standard and ipi devices, respectively. Be careful that any new
 * devices are placed within the appropriate bounded areas below.
 */

#define SELBUS_TYPE(type)          (type <= CT_SELFCUSTOM)

#define CT_DISK          0          /* SelBUS - DPIO/UDP DISC

```

```

#define CT_TAPE          1          /* SelBUS - TAPE (HI, LOW, STREAMER) */
#define CT_IOP          2          /* SelBUS - IOP */
#define CT_ETHER        3          /* SelBUS - ETHERNET */
#define CT_HSD          4          /* SelBUS - HSD */
#define CT_IPI          5          /* SelBUS - IPI */
#define CT_SELCUSTOM    6          /* SelBUS - CUSTOM CONTROLLER */

#define SEL_TYPE(type)      (CT_SELs <= type && type <= CT_SELE)

#define CT_SELs          7          /*----- standard controller fence -----*/
#define CT_8LINE        7          /* IOPBUS - 8 LINE ASYNCH */
#define CT_FLOPPY       8          /* IOPBUS - FLOPPY DISC */
#define CT_LPR          9          /* IOPBUS - LINE PRINTER */
#define CT_SCM          10         /* IOPBUS - SYNCHRONOUS COMMUNICATIONS MUX */
#define CT_IOPCUSTOM    11         /* IOPBUS - CUSTOM CONTROLLER */
#define CT_CONSOLE      12         /* IOPO - CONSOLE */
#define CT_CLOCK        13         /* - CLOCK */
#define CT_GBB          14         /* IOPBUS - GENERAL BIT/BYTE SCM */
#define CT_SELE         14         /*----- standard controller fence -----*/

#define IPI_TYPE(type)     (CT_IPIS <= type && type <= CT_IPIE)

#define CT_IPIS         15         /*----- ipi slave fence -----*/
#define CT_DIM          15         /* IPIBUS - DISC INTELLIGENT MODULE (DIM) */
#define CT_IPIE         15         /*----- ipi slave fence -----*/

/*****
 *      MAINTENANCE INTERFACE STRUCTURES AND DEFINES
 *****/
#define IM_CHECK        0          /* DEFINE TO ONE TO ALLOW DEBUG CODE */

#define IM_MAXDEV       4          /* NUMBER OF IOI MAINTENANCE DEVICES */
#define IM_NVIOCD       4          /* MAX # OF USER IOCDs IN AN IOCL */
#define IM_NPIOCD       10         /* MAX # OF PHY IOCDs AFTER MAPPING */
#define IM_MAXDCBS      4          /* MAX # OF DCBs REQUESTABLE PER MAINT.DEV */
#define IM_PXNUM        (IM_NPIOCD * IM_MAXDCBS)
#define IM_PSIZE        80         /* MAX # OF IPI COMMAND PACKET CHARACTERS */

typedef struct ioi_mstate ioi_mstateT;
typedef struct im_waitlist im_waitlistT;
typedef struct im_pagelist im_pagelistT;
typedef struct im_dcbstate im_dcbstateT;

struct ioi_mstate
{
    int      ms_pid;                /* TO PREVENT MULTIPLE OPENS BY 1 PROCESS */
    struct im_waitlist          /* MAINTENANCE STATE WAIT LIST */
    {
        short int      msw_len;          /* LENGTH OF WAITLIST */
        dcbT          *msw_dcbp[IM_MAXDCBS] /* DCBs FOR WAIT */
    } ms_waitlist;
};

```



```

struct im_pagelist          /* MAINTENANCE STATE (LOCKED) PAGE LIST */
{
    short  int    msp_cnt;          /* USAGE COUNT FOR PAGE */
    short  int    msp_page;        /* PAGE NUMBER */
} ms_pagelist[IM_PXNUM];

struct im_dcbstate         /* MAINTENANCE DCB STATE STRUCTURE */
{
    dcbT      *msd_dcbp;          /* POINTER TO DCB */
    short  int    msd_plen;        /* NUMBER OF IOCL ENTRIES */
    short  int    msd_pxlist[IM_NPIOCD]; /* IOCL ms_pagelist INDICES */
    iocdT      msd_iocl[IM_NPIOCD]; /* RESIDENT PHYSICAL IOCL */
    char      msd_packet[IM_PSIZE]; /* RESIDENT IPI CMND PACKET */
} ms_dcbstate[IM_MAXDCBS];
};

struct mnt_result          /* MAINTENANCE RESULT STRUCTURE */
{
    int          mt_proc_result;    /* RETURN FROM ioi_xxx() */
};

struct maint_sioreq       /* MAINTENANCE START I/O REQUEST STRUCTURE */
{
    iocdT      *sr_ioclp;          /* POINTER TO IOCL */
    int          sr_len;           /* NUMBER OF IOCDs */
    int          sr_errorflags;    /* ERROR PROCESSING FLAGS */
    int          sr_timeoutva;     /* TIME OUT VALUE */
};

struct maint_hioreq       /* MAINTENANCE HALT I/O REQUEST STRUCTURE */
{
    int          hr_errorflags;    /* ERROR PROCESSING FLAGS */
    int          hr_timeoutva;     /* TIME OUT VALUE */
};

struct maint_waitreq      /* MAINTENANCE WAIT REQUEST STRUCTURE */
{
    int          *wr_addrlist;     /* PTR TO LIST OF CHAN ADDRs */
    int          wr_listlen;       /* LENGTH OF ADDR LIST */
    dcbT      *wr_dcbp;           /* PLACE TO COPY UP DCB */
    caddr_t     wr_sensep;        /* PLACE TO COPY UP SENSE INFO*/
};

struct maint_commreq      /* MAINTENANCE COMM REQUEST STRUCTURE */
{
    int          cr_len;           /* LENGTH OF COMM BUFFER */
    caddr_t     cr_combuf;        /* BUF PTR TO PASS TO DRIVER */
};

```

```

struct proberesult          /* MAINTENANCE PROBE RESULT STRUCTURE */
{
    dptT          *pr_dptbase;          /* POINTER TO BASE OF DPT */
    dptT          *pr_dptrow;          /* POINTER TO ROW OF DPT */
    char          pr_devname[5];        /* DEVICE NAME */
    dcbT          *pr_dcbp;            /* POINTER TO DCB */
};

struct maint_probereq      /* MAINTENANCE PROBE REQUEST STRUCTURE */
{
    struct proberesult *pr_probefbuf;   /* RESULT BUFFER POINTER */
};

struct maint_req          /* MAINTENANCE REQUEST STRUCTURE */
{
    int            mr_devaddr;          /* CHANNEL ADDRESS */
    struct mnt_result *mr_result;       /* POINTER TO RESULT STRUCT */
    union mr_union
    {
        struct maint_sioreq mru_sr;
        struct maint_hioreq mru_hr;
        struct maint_waitreq mru_wr;
        struct maint_commreq mru_cr;
        struct maint_probereq mru_pr;
    } mr_u;
};

/*
 * DEFINES FOR CALLS TO THE DRIVER'S MAINTENANCE ROUTINE
 */
#define MR_REQUEST        0          /* REQUEST SUBCHANNEL */
#define MR_RELEASE        1          /* RELEASE SUBCHANNEL */
#define MR_COMM           2          /* COMMUNICATE TO DRIVER */

/*
 * MACRO ioi_phys():
 * The ioi_phys routine has been altered to allow two more arguments
 * on the call. This simplifies the management of IOCLs for the new
 * maintenance mode interface. To allow the old calling sequence, the
 * name of the routine was changed to ioi_physr and this macro has
 * been defined. It is used by all non-maintenance mode calls to
 * ioi_physr.
 *
 * 17 May 1984      Jonathan Bertoni of Compion      Initial Coding
 */
#define ioi_phys(viocl, piocl, lim, procp) \
    ioi_physr(viocl, piocl, lim, procp, (ioi_mstateT *) 0, (im_dcbstateT *) 0)

```



```

/*****
 *           MAINTENANCE DEBUG DEFINES           *
 *****/
#define QQ_MAXARGS      5

typedef struct
{
    unsigned char    q_routine;
    unsigned char    q_logtype;
    unsigned char    q_count;
    long             q_time;                /* SECONDS SINCE REBOOT */
    int              q_arg[QQ_MAXARGS];
} Tqq_entries;

/*
 * INDICES FOR CALLS TO QQLOG
 */
#define IOI_NOTUSED      0x00
#define IOI_PHYS        0x01
#define IOI_ICTL        0x02
#define IOI_DALLOC      0x03
#define IOI_INTR        0x04
#define IOI_TIMEOUT     0x05
#define IOI_DTIMEOUT    0x06
#define IOI_SIO         0x07
#define IOI_PGMIO       0x08
#define IOI_HIO         0x09
#define IOI_SHIO        0x0a
#define IOI_STPIO       0x0b
#define IOI_RSCTL       0x0c
#define IOI_QINTR       0x0d
#define IOI_SINTR       0x0e
#define IOI_SSENSE      0x0f
#define IOIOPEN         0x10
#define IOICLOSE        0x11
#define IOIREAD         0x12
#define IOICLEAR        0x13
#define IOIWRITE        0x14
#define IOIIOCTL        0x15
#define IOI_MINTR       0x16
#define IOI_DSTAT       0x17
#define IOI_DCBUMP      0x18
#define IOI_ISIO        0x19
#define IOI_WIO         0x1a
#define IOI_INIT        0x1b
#define I_TIS_BUSY      0x1c
#define I_TIS_HALT      0x1d
#define I_TIS_STAT      0x1e
#define IOI_IRSCTL      0x1f

#define LP_CLOSE        0x20
#define LP_INTR         0x21
#define LP_IO           0x22
#define LP_IOCTL        0x23
#define LP_OPEN         0x24
#define LP_OUTPUT       0x25

```

```

#define LP_PUTBUF          0x26
#define LP_WRITE          0x27
#define LP_STIO           0x28
#define LP_MAINT          0x29
#define LP_3spare         0x2a
#define LP_4spare         0x2b
#define LP_5spare         0x2c
#define LP_6spare         0x2d
#define LP_7spare         0x2e
#define LP_8spare         0x2f

#define CN_OPEN           0x30
#define CN_CLOSE          0x31
#define CN_READ           0x32
#define CN_WRITE          0x33
#define CN_IOCTL          0x34
#define CN_INTR           0x35
#define CN_RINT           0x36
#define CN_XINT           0x37
#define CN_PUTC           0x38
#define CN_MAINT          0x39
#define CN_INIT           0x3a
#define CN_USTART         0x3b
#define CN_PSTART         0x3c
#define CN_HANG           0x3d
#define CN_NU1            0x3e
#define CN_NU2            0x3f

#define S_IN_FATAL        0x40
#define S_LI_DUMPREAD     0x41
#define S_LI_INIT         0x42
#define S_LI_INTR         0x43
#define S_LI_PROC         0x44
#define S_LI_READ         0x45
#define S_M_QDOWN         0x46
#define S_M_QUP           0x47
#define S_QWRITE          0x48
#define S_SC_COMM         0x49
#define S_SC_DISABLE      0x4a
#define S_SC_DLBEGIN      0x4b
#define S_SC_DLDONE       0x4c
#define S_SC_INIT         0x4d
#define S_SC_INTR         0x4e
#define S_SC_MAINT        0x4f
#define S_SC_NINIT        0x50
#define S_SC_RESET        0x51
#define S_SC_RQDL         0x52
#define S_SC_STAT         0x53
#define S_SC_WAIT         0x54
#define S_SR_CONN         0x55
#define S_SR_INIT         0x56
#define S_SR_INTR         0x57
#define S_SR_TRYSTART     0x58
#define S_TD_COMACK       0x59
#define S_TD_DOWN         0x5a
#define S_TD_INIT         0x5b

```



```

#define S_TD_IODONE          0x5c
#define S_TD_REJ             0x5d
#define S_TD_TRYSEND        0x5e
#define S_TD_TSD             0x5f
#define S_TD_UP              0x60
#define S_TD_UR              0x61
#define S_TE_EXCP           0x62
#define S_TE_INIT           0x63
#define S_TE_INTR           0x64
#define S_TE_TSD            0x65
#define S_NTSCIQ            0x66
#define S_NTSCOQ            0x67
#define S_NTSCOA            0x68
#define S_NTSCOS            0x69
#define S_NTLOOQ            0x6a
#define S_SC_RSCTL           0x6b
#define S_SC_IDLE           0x6c
#define S_spare6d           0x6d
#define S_spare6e           0x6e
#define S_spare6f           0x6f

#define MT_CLOSE             0x70
#define MT_HCOMMAND         0x71
#define MT_INIT              0x72
#define MT_INTR              0x73
#define MT_IO                0x74
#define MT_IOCTL             0x75
#define MT_MAINT             0x76
#define MT_OPEN              0x77
#define MT_PHYS              0x78
#define MT_PRINT             0x79
#define MT_READ              0x7a
#define MT_RETRY             0x7b
#define MT_START             0x7c
#define MT_STRATEGY          0x7d
#define MT_WRITE             0x7e

#define S_spare7f           0x7f

#define AS_INIT              0x80
#define AS_OPEN              0x81
#define AS_CLOSE             0x82
#define AS_READ              0x83
#define AS_WRITE             0x84
#define AS_IOCTL             0x85
#define AS_INTR              0x86
#define AS_START             0x87
#define AS_STOP              0x88
#define AS_MAINT             0x89
#define AS_CMD               0x8a
#define AS_DEVREAD           0x8b
#define AS_SIGNAL            0x8c
#define AS_HALT              0x8d
#define AS_ERROR             0x8e
#define AS_I_IOCTL           0x8f
#define AS_PARAM             0x90

```

```

#define AS_SCAN                0x91
#define AS_2spare              0x92
#define AS_3spare              0x93
#define AS_4spare              0x94
#define AS_5spare              0x95
#define AS_6spare              0x96
#define AS_7spare              0x97
#define AS_8spare              0x98
#define AS_9spare              0x99

#define IM_HALTSTOPIO          0x9a
#define IM_RSCTL                0x9b
#define IM_REQUEST              0x9c
#define IM_RELEASE              0x9d
#define IM_COMM                 0x9e
#define IM_WAIT                 0x9f
#define IM_CHAN2DCBP            0xa0
#define IM_SIO                  0xa1
#define IM_PROBE                 0xa2

#define IM_SPARE3               0xa3
#define IM_SPARE4               0xa4
#define IM_SPARE5               0xa5
#define IM_SPARE6               0xa6
#define IM_SPARE7               0xa7

#define EZ_INIT                 0xa8
#define EZ_OPEN                 0xa9
#define EZ_CLOSE                0xaa
#define EZ_IOCTL                0xab
#define EZ_RESET                0xac
#define EZ_LDWCS                 0xad
#define EZ_RUN                   0xae
#define EZ_SETP                  0xaf
#define EZ_GETP                  0xb0
#define EZ_CXINTR                0xb1
#define EZ_READ                  0xb2
#define EZ_RXENQ                 0xb3
#define EZ_RXSIO                 0xb4
#define EZ_BTOM                  0xb5
#define EZ_RXINTR                0xb6
#define EZ_INPUT                 0xb7
#define EZ_WRITE                 0xb8
#define EZ_OUTPUT                0xb9
#define EZ_TXENQ                 0xba
#define EZ_TXSIO                 0xbb
#define EZ_TXINTR                0xbc
#define EZ_INTR                  0xbd
#define EZ_MAINT                 0xbe
#define EZ_STOP                  0xbf

#define M_ADJ                    0xc0
#define M_CAT                     0xc1
#define M_CLALLOC                 0xc2
#define M_COPY                     0xc3

```

```

#define M_EXPAND          0xc4
#define M_FREE           0xc5
#define M_FREEM          0xc6
#define M_GET            0xc7
#define M_GETCLR         0xc8
#define M_MORE           0xc9
#define M_PGFREE         0xca
#define M_PULLUP         0xcb
#define MBINIT           0xcc
#define M_GET_D          0xcd
#define M_CLGET_D        0xce
#define M_FREE_D         0xcf

#define HY_CHAN2HYIP     0xd0
#define HYATTACH         0xd1
#define HYRESET          0xd2
#define HYINIT           0xd3
#define HY_TIMEOUT       0xd4
#define HYSTART          0xd5
#define HYINT            0xd6
#define HY_GET_STAT      0xd7
#define HYOUTPUT         0xd8
#define HYACT            0xd9
#define HYRECVDATA       0xda
#define HYXMITDATA       0xdb
#define HYCANCEL         0xdc
#define HYPRINTDATA      0xdd
#define HYWATCH          0xde
#define HYLOG            0xdf
#define HYIOCTL          0xe0

#define HS_INIT          0xe1
#define HS_INTR          0xe2
#define HS_MAINT         0xe3
#define HS_STOP          0xe4
#define HS_SIO           0xe5
#define HS_HIO           0xe6
#define HS_EI            0xe7
#define HS_DI            0xe8
#define HS_DAI           0xe9
#define HS_TD            0xea
#define HS_OPEN          0xeb
#define HS_CLOSE         0xec
#define HS_READ          0xed
#define HS_WRITE         0xee
#define HS_IOCTL         0xef

#define IF_BTOM          0xf0
#define IF_MTOB          0xf1

```



```

/*
 * reasons
 */
#define LOG_ENTER          0x0
#define LOG_EXIT          0x1
#define LOG_SIO           0x2
#define LOG_SHIO         0x3
#define LOG_RSCTL        0x4
#define LOG_OPOINT       0x5
#define LOG_1POINT       0x6
#define LOG_2POINT       0x7
#define LOG_3POINT       0      x8
#define LOG_TIO 0        x9
#define LOG_SLEEP        0xa
#define LOG_WAKEUP       0xb

#ifdef ILOG_STRINGS
char *q_rtn_names[] =
{
    "IOI_NOTUSED",          /* 0x00 */
    "IOI_PHYS",            /* 0x01 */
    "IOI_ICTL",            /* 0x02 */
    "IOI_DALLOC",          /* 0x03 */
    "IOI_INTR",            /* 0x04 */
    "IOI_TIMEOUT",        /* 0x05 */
    "IOI_DTIMEOUT ",      /* 0x06 */
    "IOI_SIO",              /* 0x07 */
    "IOI_PGMIO ",         /* 0x08 */
    "IOI_HIO",              /* 0x09 */
    "IOI_SHIO",            /* 0x0a */
    "IOI_STPIO",           /* 0x0b */
    "IOI_RSCTL",           /* 0x0c */
    "IOI_QINTR",           /* 0x0d */
    "IOI_SINTR",           /* 0x0e */
    "IOI_SSSENSE",         /* 0x0f */
    "IOIOPEN",             /* 0x10 */
    "IOICLOSE",            /* 0x11 */
    "IOIREAD",             /* 0x12 */
    "IOICLEAR ",          /* 0x13 */
    "IOIWRITE",            /* 0x14 */
    "IOIIOCTL ",          /* 0x15 */
    "IOI_MINTR",           /* 0x16 */
    "IOI_DSTAT ",         /* 0x17 */
    "IOI_DCBDUMP ",       /* 0x18 */
    "IOI_ISIO",            /* 0x19 */
    "IOI_WIO",              /* 0x1a */
    "IOI_INIT",            /* 0x1b */
    "I_TIS_BUSY",          /* 0x1c */
    "I_TIS_HALT",          /* 0x1d */
    "I_TIS_STAT",          /* 0x1e */
    "IOI_IRSCTL",          /* 0x1f */

    "LP_CLOSE",            /* 0x20 */
    "LP_INTR",             /* 0x21 */
    "LP_IO",                /* 0x22 */
    "LP_IOCTL",            /* 0x23 */

```

```

"LP_OPEN",           /* 0x24 */
"LP_OUTPUT",        /* 0x25 */
"LP_PUTBUF",        /* 0x26 */
"LP_WRITE",         /* 0x27 */
"LP_STIO",          /* 0x28 */
"LP_MAINT",         /* 0x29 */
"LP_3spare",        /* 0x2a */
"LP_4spare",        /* 0x2b */
"LP_5spare",        /* 0x2c */
"LP_6spare",        /* 0x2d */
"LP_7spare",        /* 0x2e */
"LP_8spare",        /* 0x2f */

"CN_OPEN",          /* 0x30 */
"CN_CLOSE",         /* 0x31 */
"CN_READ",          /* 0x32 */
"CN_WRITE",         /* 0x33 */
"CN_IOCTL",         /* 0x34 */
"CN_INTR",          /* 0x35 */
"CN_RINT",          /* 0x36 */
"CN_XINT",          /* 0x37 */
"CN_PUTC",          /* 0x38 */
"CN_MAINT",         /* 0x39 */
"CN_INIT",          /* 0x3a */
"CN_USTART",        /* 0x3b */
"CN_PSTART",        /* 0x3c */
"CN_HANG",          /* 0x3d */
"CN_NU1",           /* 0x3e */
"CN_NU2",           /* 0x3f */

"S_IN_FATAL",       /* 0x40 */
"S_LI_DUMPREAD",   /* 0x41 */
"S_LI_INIT",        /* 0x42 */
"S_LI_INTR",        /* 0x43 */
"S_LI_PROC",        /* 0x44 */
"S_LI_READ",        /* 0x45 */
"S_M_QDOWN",        /* 0x46 */
"S_M_QUP",          /* 0x47 */
"S_QWRITE",         /* 0x48 */
"S_SC_COMM",        /* 0x49 */
"S_SC_DISABLE",    /* 0x4a */
"S_SC_DLBEGIN",    /* 0x4b */
"S_SC_DLDONE",     /* 0x4c */
"S_SC_INIT",        /* 0x4d */
"S_SC_INTR",        /* 0x4e */
"S_SC_MAINT",       /* 0x4f */
"S_SC_NINIT",       /* 0x50 */
"S_SC_RESET",       /* 0x51 */
"S_SC_RQDL",        /* 0x52 */
"S_SC_STAT",        /* 0x53 */
"S_SC_WAIT",        /* 0x54 */
"S_SR_CONN",        /* 0x55 */
"S_SR_INIT",        /* 0x56 */
"S_SR_INTR",        /* 0x57 */
"S_SR_TRYSTART",   /* 0x58 */
"S_TD_COMACK",     /* 0x59 */

```

```

"S_TD_DOWN",           /* 0x5a */
"S_TD_INIT",          /* 0x5b */
"S_TD_IODONE",        /* 0x5c */
"S_TD_REJ",           /* 0x5d */
"S_TD_TRYSEND", /* 0x5e */
"S_TD_TSD",           /* 0x5f */
"S_TD_UP",            /* 0x60 */
"S_TD_UR",            /* 0x61 */
"S_TE_EXCP",          /* 0x62 */
"S_TE_INIT",          /* 0x63 */
"S_TE_INTR",          /* 0x64 */
"S_TE_TSD",           /* 0x65 */
"S_NTSCIQ",           /* 0x66 */
"S_NTSCOQ",           /* 0x67 */
"S_NTSCOA",           /* 0x68 */
"S_NTSCOS",           /* 0x69 */
"S_NTLOOQ",           /* 0x6a */
"S_spare6b",          /* 0x6b */
"S_spare6c",          /* 0x6c */
"S_spare6d",          /* 0x6d */
"S_spare6e",          /* 0x6e */
"S_spare6f",          /* 0x6f */

"MT_CLOSE",           /* 0x70 */
"MT_HCOMMAND",        /* 0x71 */
"MT_INIT",            /* 0x72 */
"MT_INTR",            /* 0x73 */
"MT_IO",              /* 0x74 */
"MT_IOCTL",           /* 0x75 */
"MT_MAINT",           /* 0x76 */
"MT_OPEN",            /* 0x77 */
"MT_PHYS",            /* 0x78 */
"MT_PRINT",           /* 0x79 */
"MT_READ",            /* 0x7a */
"MT_RETRY",           /* 0x7b */
"MT_START",           /* 0x7c */
"MT_STRATEGY",        /* 0x7d */
"MT_WRITE",           /* 0x7e */

"S_spare7f",          /* 0x7f */

"AS_INIT",            /* 0x80 */
"AS_OPEN",            /* 0x81 */
"AS_CLOSE",           /* 0x82 */
"AS_READ",            /* 0x83 */
"AS_WRITE",           /* 0x84 */
"AS_IOCTL",           /* 0x85 */
"AS_INTR",            /* 0x86 */
"AS_START",           /* 0x87 */
"AS_STOP",            /* 0x88 */
"AS_MAINT",           /* 0x89 */
"AS_CMD",             /* 0x8a */
"AS_DEVREAD",         /* 0x8b */
"AS_SIGNAL",          /* 0x8c */
"AS_HALT",            /* 0x8d */
"AS_ERROR",           /* 0x8e */

```



```

"AS_I_IOCTL",          /* 0x8f */
"AS_PARAM",           /* 0x90 */
"AS_SCAN",            /* 0x91 */
"AS_2SPARE",          /* 0x92 */
"AS_3SPARE",          /* 0x93 */
"AS_4SPARE",          /* 0x94 */
"AS_5SPARE",          /* 0x95 */
"AS_6SPARE",          /* 0x96 */
"AS_7SPARE",          /* 0x97 */
"AS_8SPARE",          /* 0x98 */
"AS_9SPARE",          /* 0x99 */

"IM_HALTSTOPIO",      /* 0x9a */
"IM_RSCTL",           /* 0x9b */
"IM_REQUEST",         /* 0x9c */
"IM_RELEASE",         /* 0x9d */
"IM_COMM",            /* 0x9e */
"IM_WAIT",            /* 0x9f */
"IM_CHAN2DCBP", /* 0xa0 */
"IM_SIO",             /* 0xa1 */
"IM_PROBE",           /* 0xa2 */

"IM_SPARE3",          /* 0xa3 */
"IM_SPARE4",          /* 0xa4 */
"IM_SPARE5",          /* 0xa5 */
"IM_SPARE6",          /* 0xa6 */
"IM_SPARE7",          /* 0xa7 */

"EZ_INIT",            /* 0xa8 */
"EZ_OPEN",            /* 0xa9 */
"EZ_CLOSE",           /* 0xaa */
"EZ_IOCTL",           /* 0xab */
"EZ_RESET",           /* 0xac */
"EZ_LDWCS",           /* 0xad */
"EZ_RUN",             /* 0xae */
"EZ_SETP",            /* 0xaf */
"EZ_GETP",            /* 0xb0 */
"EZ_CXINTR",          /* 0xb1 */
"EZ_READ",            /* 0xb2 */
"EZ_RXENQ",           /* 0xb3 */
"EZ_RXSIO",           /* 0xb4 */
"EZ_BTOM",            /* 0xb5 */
"EZ_RXINTR",          /* 0xb6 */
"EZ_INPUT",           /* 0xb7 */
"EZ_WRITE",           /* 0xb8 */
"EZ_OUTPUT",          /* 0xb9 */
"EZ_TXENQ",           /* 0xba */
"EZ_TXSIO",           /* 0xbb */
"EZ_TXINTR",          /* 0xbc */
"EZ_INTR",            /* 0xbd */
"EZ_MAINT",           /* 0xbe */
"EZ_STOP",            /* 0xbf */

"M_ADJ",              /* 0xc0 */

```

```

"M_CAT", /* 0xc1 */
"M_CLALLOC", /* 0xc2 */
"M_COPY", /* 0xc3 */
"M_EXPAND", /* 0xc4 */
"M_FREE", /* 0xc5 */
"M_FREEM", /* 0xc6 */
"M_GET", /* 0xc7 */
"M_GETCLR", /* 0xc8 */
"M_MORE", /* 0xc9 */
"M_PGFREE", /* 0xca */
"M_PULLUP", /* 0xcb */
"MBINIT", /* 0xcc */
"M_GET_D", /* 0xcd */
"M_CLGET_D", /* 0xce */
"M_FREE_D", /* 0xcf */

"HY_CHAN2HYIP", /* 0xd0 */
"HYATTACH", /* 0xd1 */
"HYRESET", /* 0xd2 */
"HYINIT", /* 0xd3 */
"HY_TIMEOUT", /* 0xd4 */
"HYSTART", /* 0xd5 */
"HYINT", /* 0xd6 */
"HY_GET_STAT", /* 0xd7 */
"HYOUTPUT", /* 0xd8 */
"HYACT", /* 0xd9 */
"HYRECVDATA", /* 0xda */
"HYXMITDATA", /* 0xdb */
"HYCANCEL", /* 0xdc */
"HYPRINTDATA", /* 0xdd */
"HYWATCH", /* 0xde */
"HYLOG", /* 0xdf */
"HYIOCTL", /* 0xe0 */

"HS_INIT", /* 0xe1 */
"HS_INTR", /* 0xe2 */
"HS_MAINT", /* 0xe3 */
"HS_STOP", /* 0xe4 */
"HS_SIO", /* 0xe5 */
"HS_HIO", /* 0xe6 */
"HS_EI", /* 0xe7 */
"HS_DI", /* 0xe8 */
"HS_DAI", /* 0xe9 */
"HS_TD", /* 0xea */
"HS_OPEN", /* 0xeb */
"HS_CLOSE", /* 0xec */
"HS_READ", /* 0xed */
"HS_WRITE", /* 0xee */
"HS_IOCTL", /* 0xef */

"IF_BTOM", /* 0xf0 */
"IF_MTOB", /* 0xf1 */
};

char *q_call_types[] =
{

```

```

"LOG_ENTER",          /* 0x0 */
"LOG_EXIT",           /* 0x1 */
"LOG_SIO",            /* 0x2 */
"LOG_SHIO",           /* 0x3 */
"LOG_RSCTL",          /* 0x4 */
"LOG_OPOINT",         /* 0x5 */
"LOG_1POINT",         /* 0x6 */
"LOG_2POINT",         /* 0x7 */
"LOG_3POINT",         /* 0x8 */
"LOG_TIO",            /* 0x9 */
"LOG_SLEEP",          /* 0xa */
"LOG_WAKEUP"         /* 0xb */
};
#endif ILOG_STRINGS

#ifdef QQDEBUG

#define QQLOG(X) qqlog X

#else QQDEBUG

#define QQLOG(X) /* null */
#define qqoff() /* null */
#define qqon() /* null */

#endif QQDEBUG

#ifdef DEBUG

extern int Ioi_debug;
#define PRINTF(X) if (Ioi_debug) printf X
#define DCBDUMP(X) ioi_dcbdump X
#define DSTAT(X) ioi_dstat X

#else DEBUG

#define PRINTF(X) /* null */
#define DCBDUMP(X) /* null */
#define DSTAT(X) /* null */

#endif DEBUG

```



```

/*****
 *
 *          EXTENDED I/O INSTRUCTION CONDITION CODES
 *
 *****/
#define CC_ACT_ECHO      (0x0)      /* (00) REQUEST ACTIVATE, WILL ECHO */
#define CC_BUSY         (0x1)      /* (08) CHANNEL BUSY */
#define CC_INOP         (0x2)      /* (10) CHANNEL INOPERABLE */
#define CC_SUBBUSY      (0x3)      /* (18) SUBCHANNEL BUSY */
#define CC_SSTORED      (0x4)      /* (20) STATUS STORED */
#define CC_UNSUPPORTED  (0x5)      /* (28) UNSUPPORTED TRANSACTION */
#define CC_UA_6         (0x6)      /* (30) UNASSIGNED */
#define CC_UA_7         (0x7)      /* (38) UNASSIGNED */
#define CC_REQ_ACC      (0x8)      /* (40) REQUEST ACCEPTED, NO ECHO */
#define CC_UA_9         (0x9)      /* (48) UNASSIGNED */
#define CC_UA_A         (0xa)      /* (50) UNASSIGNED */
#define CC_UA_B         (0xb)      /* (58) UNASSIGNED */
#define CC_UA_C         (0xc)      /* (60) UNASSIGNED */
#define CC_UA_D         (0xd)      /* (68) UNASSIGNED */
#define CC_UA_E         (0xe)      /* (70) UNASSIGNED */
#define CC_UA_F         (0xf)      /* (78) UNASSIGNED

/*****
 *
 *          iocd_optypes[] DEFINES
 *
 *****/
#define T_NULL  0x00      /* NOTHING */
#define T_MEM   0x01      /* MEMORY REFERENCE OPCODE */
#define T_MEMRD 0x02      /* READS MEMORY */
#define T_MEMWR 0x04      /* WRITES MEMORY */
#define T_VAL   0x80      /* VALID FLAG */
#define T_DC    0x40      /* DATA CHAINABLE OPCODE

#define T_CCTL  (T_MEM|T_MEMRD| T_VAL) /* CHANNEL CONTROL */
#define T_SENS  (T_MEM|T_MEMWR| T_VAL) /* SENSE */
#define T_TIC   (T_NULL)      /* TRANSFER IN CHANNEL */
#define T_RDR   (T_MEM|T_MEMWR|T_DC|T_VAL) /* READ REVERSE */
#define T_WR    (T_MEM|T_MEMRD|T_DC|T_VAL) /* WRITE DATA */
#define T_RD    (T_MEM|T_MEMWR|T_DC|T_VAL) /* READ DATA */
#define T_CTL   (T_MEM|T_MEMRD| T_VAL) /* DEVICE CONTROL

#define IS_MEM_REF(op)  ( op & T_MEM ) /* OPCODE IS MEMEMORY REF TYPE */
#define IS_DCHAINABLE(op) ( op & T_DC ) /* OPCODE IS DATA CHAINABLE */
#define IS_VAL(op)      ( op & T_VAL ) /* OPCODE IS SUPPORTED

/*****
 *
 *          RETURN VALUES FROM ioi_sio(), ioi_hio(), etc
 *
 *****/
#define IS_OK      0      /* I/O STARTED SUCCESSFULLY */
#define IS_BY      1      /* I/O FAILED - BUSY */
#define IS_IP      2      /* I/O FAILED - INTERRUPT PENDING */
#define IS_RTCNT   3      /* I/O FAILED - TOO MANY BUSY RETRIES */
#define IS_BADSTATE 4      /* IOI IN WRONG STATE FOR OPERATION */
#define IS_NOCTL   5      /* (INTERNAL USE) - DEVICE INOPERABLE */
#define IS_TRYING  -1     /* (INTERNAL USE) - LOOP CONTROL

```

```

* VALUES PASSED AS "reason" TO DRIVER'S INTERRUPT SERVICE ROUTINE *
*****/
#define ICS_OK          0          /* SUCCESSFUL I/O COMPLETION */
#define ICS_AB          1          /* I/O COMPLETION WITH ABNORMAL STATUS */
#define ICS_SI          2          /* SPONTANEOUS INTERRUPT */
#define ICS_HIO         3          /* I/O COMPLETION BY HIO OR TIME OUT */
#define ICS_RSCTL       4          /* I/O CANCELLED VIA RSCTL AFTER HIO */
#define ICS_BROKEN      5          /* DEVICE FAILED TO ACCEPT HIO or RSCTL*/

#ifdef ICS_STRINGS

char *Ioi_icsreasons[] =
{
    "ICS_OK      ",
    "ICS_AB      ",
    "ICS_SI      ",
    "ICS_HIO     ",
    "ICS_RSCTL   ",
    "ICS_BROKEN",
};

#endif ICS_STRINGS

/*****
* Flags passed to ioi_phys() in the "junk" field of the proto IOCL *
*****/
#define PIO_WRITE      0x00
#define PIO_READ       0x01
#define PIO_UAREA      0x02
#define PIO_DIRTY      0x04

#endif Hioi
/*
* (c) Copyright 1986 Gould Inc.
* All Rights Reserved.
*/

```


References

Gould Inc. *CONCEPT 32/67 CPU and Gould SS6/CMOS CPU (Real-Time) Reference Manual*. Publication Order Number 301-000410.

Gould Inc. *Gould CONCEPT 32/97 Computer Basic System Reference Manual*. Publication Order Number 301-003070.

Gould Inc. *High-speed Data Interface, Model 9130/ High-speed Data Interface II, Model 9131/ High-speed Data Interface, Model 9132/ High-speed Data Inter-bus Link II, Model 9135/ High-speed Data Inter-bus Link, Model 9136 Technical Manual*. Publication Order Number 303-000270.

Users Group Membership Application

USER ORGANIZATION: _____

REPRESENTATIVE(S): _____

ADDRESS: _____

TELEX NUMBER: _____ PHONE NUMBER: _____

NUMBER AND TYPE OF GOULD CSD COMPUTERS: _____

OPERATING SYSTEM AND REV. LEVEL: _____

APPLICATIONS (Please Indicate)

- | | | |
|--|---|---|
| <p>1. EDP</p> <ul style="list-style-type: none">A. Inventory ControlB. Engineering & Production Data ControlC. Large Machine Off-LoadD. Remote Batch TerminalE. Other | <p>2. Communications</p> <ul style="list-style-type: none">A. Telephone System MonitoringB. Front End ProcessorsC. Message SwitchingD. Other | <p>3. Design & Drafting</p> <ul style="list-style-type: none">A. ElectricalB. MechanicalC. ArchitecturalD. CartographyE. Image ProcessingF. Other |
| <p>4. Industrial Automation</p> <ul style="list-style-type: none">A. Continuous Process Control Op.B. Production Scheduling & ControlC. Process PlanningD. Numerical ControlE. Other | <p>5. Laboratory and Computational</p> <ul style="list-style-type: none">A. SeismicB. Scientific CalculationC. Experiment MonitoringD. Mathematical ModelingE. Signal ProcessingF. Other | <p>6. Energy Monitoring & Control</p> <ul style="list-style-type: none">A. Power GenerationB. Power DistributionC. Environmental ControlD. Meter MonitoringE. Other |
| <p>7. Simulation</p> <ul style="list-style-type: none">A. Flight SimulatorsB. Power Plant SimulatorsC. Electronic WarfareD. Other | <p>8. Other</p> | <p>Please return to:</p> <p>Users Group Representative</p> <p>Date: _____</p> |

Gould Inc., Computer Systems Division Users Group . . .

The purpose of the Gould CSD Users Group is to help create better User/User and User/Gould CSD communications.

There is no fee to join the Users Group. Simply complete the Membership Application on the reverse side and mail to the Users Group Representative. You will automatically receive Users Group Newsletters, Referral Guide and other pertinent Users Group activity information.

Fold and Staple for Mailing



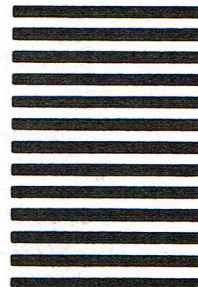
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 947 FT. LAUDERDALE, FL

POSTAGE WILL BE PAID BY ADDRESSEE

GOULD INC., COMPUTER SYSTEMS DIVISION
ATTENTION: USERS GROUP REPRESENTATIVE
6901 W. SUNRISE BLVD.
P. O. BOX 409148
FT. LAUDERDALE FL 33340-9970



(Detach Here)



Fold and Staple for Mailing

