GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX™86 AND iRMX™88 I/O SYSTEMS

Order Number: 142926-003

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

REV.	REVISION HISTORY	PRINT DATE
-001	Original Issue	11/80
-002	Updated to reflect the changes in version 3.0 of the iRMX 86 software.	5/81
-003	Broadened to cover the iRMX 88 Executive and reorganized for improved usability.	12/81

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

PREFACE

The I/O System is the part of the iRMX 86 Operating System and the iRMX 88 Real-Time Multitasking Executive that provides you with the capability of accessing files on peripheral devices. Each of these I/O Systems is implemented as a set of file drivers and a set of device drivers. A file driver provides user access to a particular type of file, independent of the device on which the file resides. A device driver provides a standard interface between a particular device and one or more file drivers. Thus, by adding device drivers, your application system can support additional types of devices. And it can do this without changing the user interface, since the file drivers remain unchanged.

This manual describes how to write device drivers to interface with the I/O Systems. It illustrates the basic concepts of device drivers and describes the different types of device drivers (common, random access, and custom).

READER LEVEL

This manual assumes that you are a systems-level programmer experienced in dealing with I/O devices. In particular, it assumes that you are familiar with the following:

- The PL/M-86 programming language and/or the MCS-86 Macro Assembly Language.
- The hardware codes necessary to perform actual read and write operations on your I/O device. This manual does not document these device-dependent instructions.

If you plan to write a device driver that uses iRMX 86 system calls, you should be familiar with the following, as well:

- The iRMX 86 Operating System and the concepts of tasks, segments, and other objects.
- The I/O System, as described in the iRMX 86 Basic I/O SYSTEM REFERENCE MANUAL. This manual documents the user interface to the I/O System.
- Regions, as described in the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

And if you plan to write a device driver that uses iRMX 88 functions or system calls, you should be familiar with the iRMX 88 Reference Manual.

RELATED PUBLICATIONS

The following manuals provide additional information that may be helpful to users of this manual.

Manual	Number
iRMX™ 86 Nucleus Reference Manual	9803122
iRMX™ 86 Basic I/O System Reference Manual	9803123
iRMX™ 86 Extended I/O System Reference Manual	143308
iRMX™ 86 Loader Reference Manual	143381
iRMX™ 86 System Programmer's Reference Manual	142721
iRMX™ 86 Configuration Guide	9803126
iRMX™ 88 Reference Manual	143232
iRMX™ 80/88 Interactive Configuration Utility User's Guide	142603
PL/M-86 Programming Manual for 8080/8085-Based Development Systems	9800466
PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems	9800478
PL/M-86 User's Guide for 8086-Based Development Systems	121636
8086/8087/8088 Macro Assembly Language Reference Manual for 8080/8085-Based Development Systems	121623
8086/8087/8088 Macro Assembly Language Reference Manual for 8086-Based Development Systems	121627
8086/8087/8088 Macro Assembler Operating Instructions for 8080/8085-Based Development Systems	121624
8086/8087/8088 Macro Assembler Operating Instructions for 8086-Based Development Systems	121628
8086 Family Utilities User's Guide for 8080/8085-Based Development Systems	9800639
iAPX 86 Family Utilities User's Guide for 8086-Based Development Systems	121616

CONTENTS

FIGURES

TABLES

CHAPTER 1. INTRODUCTION

The iRMX 86 and iRMX 88 I/O Systems are each implemented as a set of file drivers and a set of device drivers. File drivers provide the support for particular types of files (for example, the named file driver provides the support needed in order to use named files). Device drivers provide the support for particular devices (for example, an iSBC 215 device driver provides the facilities that enable an iSBC 215 Winchester drive to be used with the I/O System). Each type of file has its own file driver and each device has its own device driver.

One of the reasons that the I/O Systems are broken up in this manner is to provide device-independent I/O. Application tasks communicate with file drivers, not with device drivers. This allows tasks to manipulate all files in the same manner, regardless of the devices on which they reside. File drivers, in turn, communicate with device drivers, which provide the instructions necessary to manipulate physical devices. Figure 1-1 shows these levels of communication.

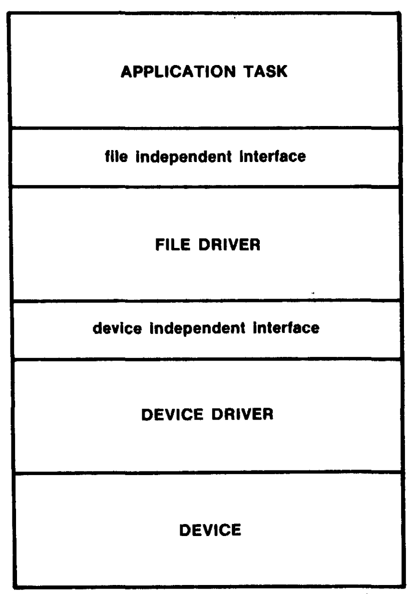

Figure 1-1. Communication Levels

The I/O System provides a standard interface between file drivers and device drivers. To a file driver, a device is merely a standard block of data in a table. In order to manipulate a device, the file driver calls the device driver procedures listed in the table. To a device driver, all file drivers seem the same. Every file driver calls device drivers in the same manner. This means that the device driver does not need to concern itself with the concept of a file driver. It sees itself as being called by the I/O System and it returns information to the I/O System. This standard interface has the following advantages:

- The hardware configuration can be changed without extensive modifications to the software. Instead of modifying entire file drivers when you want to change devices, you need only substitute a different device driver and modify the table.

- The I/O System can support a greater range of devices. It can support any device as long as you can provide for the device a driver that interfaces to the file drivers in the standard manner.

I/O DEVICES AND DEVICE DRIVERS

Each I/O device consists of a controller and one or more units. A device as a whole is identified by a device number. Units are identified by unit number and device-unit number. The unit number identifies the unit within the device and the device-unit number identifies the unit among all the units of all the devices. Figure 1-2 contains a simplified drawing of three I/O devices and their device, unit, and device-unit numbers.

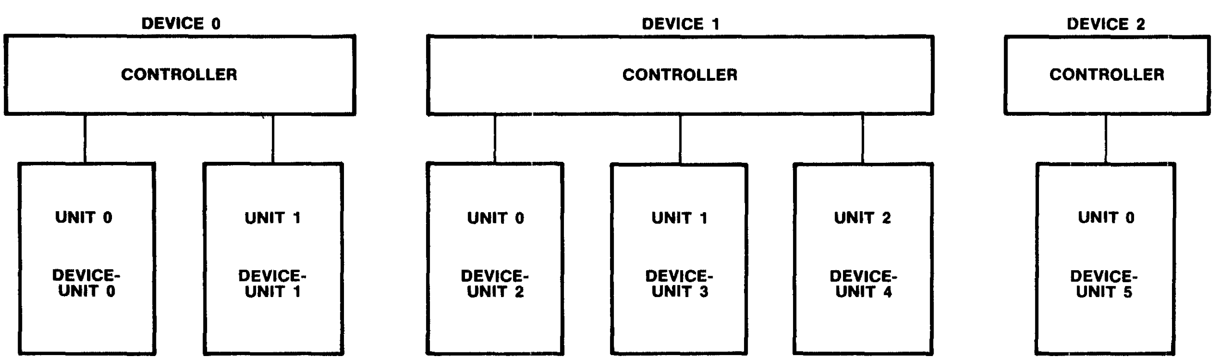

Figure 1-2. Device Numbering

You must provide a device driver for every device in your hardware configuration. That device driver must handle the I/O requests for all of the units the device supports. Different devices can use different device drivers; or if they are the same kind of device, they can share the same device driver code. (For example, two iSBC 215 controllers are two separate devices and each has its own device driver. However, these device drivers share common code.)

I/O REQUESTS

To the device driver, an I/O request is a request by the I/O System for the device to perform a certain operation. Operations supported by the I/O System are:

- Read
- Write
- Seek
- Special
- Attach device
- Detach device
- Open
- Close

The I/O System makes an I/O request by sending an I/O request/result segment (IORS) containing the necessary information to the device driver. (The IORS is described in Chapter 2.) The device driver must translate this request into commands to the device in order to cause the device to perform the requested operation.

TYPES OF DEVICE DRIVERS

The I/O System supports three types of device drivers: custom, common, and random access. A custom device driver is one that the user creates in its entirety. This type of device driver may assume any form and may provide any functions that the user wishes, as long as the I/O System can access it by calling four procedures, designated as Initialize I/O, Finish I/O, Queue I/O, and Cancel I/O.

The I/O System provides the basic support routines for the common and random access device driver types. These support routines provide a queueing mechanism, an interrupt handler, and other features needed by common or random access devices. If your device fits into the common or random access device classification, you need to write only the specialized, device-dependent procedures and interface them to the ones provided by the I/O System in order to create a complete device driver.

HOW TO READ THIS MANUAL

This manual is for people who plan to write device drivers for use with iRMX 86- and/or iRMX 88-based systems. Because there are numerous terminology differences between the two iRMX systems, the tone of this manual is general, unlike that of other manuals for either system. For iRMX 88 users, this should not be a problem, but iRMX 86 users should take note of the following:

- In a number of places the phrase "the location of" is substituted for "a token for".
- The "device data storage area" that is alluded to in many places is actually an iRMX 86 segment.
- The term "resources" usually means "objects". It is clear from context which meaning is intended for the word "resources".

CHAPTER 2. DEVICE DRIVER INTERFACES

Because a device driver is a collection of software routines that manages a device at a basic level, it must transform general instructions from the I/O System into device-specific instructions which it then sends to the device itself. Thus a device driver has two types of interfaces: an interface to the I/O System, which is the same for all device drivers, and an interface to the device itself, which varies according to device. This chapter discusses these interfaces.

I/O SYSTEM INTERFACES

The interface between the device driver and the I/O System consists of two data structures, the device-unit information block (DUIB) and the I/O request/result segment (IORS).

DEVICE-UNIT INFORMATION BLOCK (DUIB)

The DUIB is an interface between a device driver and the I/O System in the sense that the DUIB contains the addresses of the device driver routines. By accessing the DUIB for a unit, the I/O System can call the appropriate device driver. All devices, no matter how diverse, use this standard interface to the I/O System. You must provide a DUIB for each device-unit in your hardware system. You supply the information for your DUIBs as part of the configuration process.

DUIB Structure

The structure of the DUIB is defined as follows:

```
DECLARE
    DEV$UNIT$INFO$BLOCK  STRUCTURE(
        NAME(14)        BYTE,
        FILE$DRIVERS    WORD,
        FUNCTS          BYTE,
        FLAGS           BYTE,
        DEV$GRAN        WORD,
        LOW$DEV$SIZE    WORD,
        HIGH$DEV$SIZE   WORD,
        DEVICE          BYTE,
        UNIT            BYTE,
        DEV$UNIT        WORD,
        INIT$IO         WORD,
        FINISH$IO       WORD,
        QUEUE$IO        WORD,
        CANCEL$IO       WORD,
        DEVICE$INFO$P   POINTER,
        UNIT$INFO$P     POINTER,
        UPDATE$TIMEOUT  WORD,
        NUM$BUFFERS     WORD,
        PRIORITY        BYTE);
```

where:

NAME — BYTE array specifying the name of the DUIB. This name uniquely identifies the device-unit to the I/O System. If you are an iRMX 86 user, you specify this name when attaching a unit by means of the RQAPHYSICAL$ATTACH$DEVICE system call. If you are an iRMX 88 user, you specify the name when configuring with the Interactive Configuration Utility. Device drivers can ignore this field.

FILE$DRIVERS — WORD specifying file driver validity. Setting bit number i of this word implies that file driver number i+1 can attach this device-unit. Clearing bit number i implies that file driver i+1 cannot attach this device-unit. The low-order bit is bit 0. The bits are associated with the file drivers as follows:

bit	file driver
0	physical (no. 1)
1	stream (no. 2)
3	named (no. 4)

The remainder of the word must be set to zero. (For iRMX 88 users, the physical file driver can attach devices which are files, and the named file driver can attach devices which can contain multiple files.) Device drivers can ignore this field.

FUNCTS

BYTE specifying the I/O function validity for this device-unit. Setting bit number i implies that this device-unit supports function number i. Clearing bit number i implies that the device-unit does not support function number i. The low-order bit is bit 0. The bits are associated with the functions as follows:

bit	function
0	F$READ
1	F$WRITE
2	F$SEEK
3	F$SPECIAL
4	F$ATTACH$DEV
5	F$DETACH$DEV
6	F$OPEN
7	F$CLOSE

Bits 4 and 5 should always be set. Every device driver requires these functions.

This field is used for informational purposes only. The setting or clearing of bits in this field does not limit the device driver from performing any I/O function. In fact, each device driver must be able to support all of the I/O functions, either by performing the function or by returning a condition code indicating the inability of the device to perform that function. However, in order to provide accurate status information, this field should indicate the device's ability to perform the I/O functions.

FLAGS

BYTE specifying characteristics of diskette devices. The significance of the bits is as follows:

bit	meaning
0	0 = not a diskette device; 1 = diskette device
1	0 = single density; 1 = double density
2	0 = single sided; 1 = double sided
3	0 = 8-inch diskettes; 1 = 5 1/4-inch diskettes
4-7	reserved

For non-diskette devices, bits 1-7 of this field have no significance.

DEV$GRAN

WORD specifying the device granularity in bytes. This parameter is most important for random access devices. It specifies the minimum number of bytes of information that the device reads or writes in one operation. You should set this value equal to the volume granularity specified when the volume was formatted.

LOWDEVSIZE HIGHDEVSIZE	WORD pair that forms a 32-bit field specifying the number of bytes of information that the device-unit can store.
DEVICE	BYTE specifying the device number of the device with which this device-unit is associated. Device drivers can ignore this field.
UNIT	BYTE specifying the unit number of this device-unit. This distinguishes the unit from the other units of the device.
DEV$UNIT	WORD specifying the device-unit number. This number distinguishes the device-unit from the other units in the entire hardware system. Device drivers can ignore this field.
INIT$IO	WORD specifying the offset in the code segment of this unit's Initialize I/O device driver procedure.
FINISH$IO	WORD specifying the offset in the code segment of this unit's Finish I/O device driver procedure.
QUEUE$IO	WORD specifying the offset in the code segment of this unit's Queue I/O device driver procedure.
CANCEL$IO	WORD specifying the offset in the code segment of this unit's Cancel I/O device driver procedure.
DEVICE$INFO$P	POINTER to a structure which contains additional information about the device. The common and random access device drivers require device information structures of a particular format. These structures are described in Chapter 4. If you are writing a custom driver, you can place information in this structure depending on the needs of your driver. Specify a zero for this parameter if the associated device driver does not use this field.
UNIT$INFO$P	POINTER to a structure that contains additional information about the unit. Random access device drivers require this unit information structure in a particular format. Refer to Chapter 4 for further information. If you are writing a custom device driver, place information in this structure depending on the needs of your driver. Specify a zero for this parameter if the associated device driver does not use this field.
UPDATE$TIMEOUT	WORD specifying the number of system time units that the I/O System is to wait before writing a partial sector after processing a write request for a disk device. In the case of drivers for non-disk devices, this field should be set to OFFFFH during configuration. Device drivers can ignore this field.

NUM$BUFFERS — WORD which, if not zero, specifies that the device is of the random access variety and indicates the number of buffers I/O System may allocate. The I/O System uses these buffers to perform data blocking and deblocking operations. That is, it guarantees that data is read or written beginning on sector boundaries. If you desire, the random access support routines can also be made to guarantee that no data is written or read across track boundaries in a single request (see the section on the unit information table in Chapter 4). A value of zero indicates that the device is not a random access device. Device drivers can ignore this field.

PRIORITY — BYTE specifying the priority of the I/O System service task for the device. Device drivers can ignore this field.

Using the DUIBs

In order to use the I/O System to connect your application software and any files on a device-unit, the unit must first be attached. If you are an iRMX 88 user, this is done automatically when you first attach or create a file on the unit. If you are an iRMX 86 user, you attach the unit by using the RQAPHYSICAL$ATTACH$DEVICE system call (refer to the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL for a description of this system call).

When you cause a unit to become attached, the I/O System assumes that the device-unit identified by the device name field of the DUIB has the characteristics identified in the remainder of the DUIB. Thus, whenever the application software makes any I/O requests using the connection to the attached device-unit, the I/O System ascertains the characteristics of that unit, including which device driver procedures to call in order to actually process the I/O request, by means of the associated DUIB. The I/O System looks at the DUIB and calls the appropriate device driver routine listed there in order to process the I/O request.

If you would like the I/O System to assume different characteristics at different times for a particular device-unit, you can accomplish this by providing alternate DUIBs for the device-unit. If you supply multiple DUIBs, each containing identical device number, unit number, and device-unit number parameters, but different DUIB device name parameters, you can choose which DUIB to associate with a device-unit that you are attaching by specifying the appropriate dev$name parameter in the RQ$A$PHYSICAL$ATTACH$DEVICE system call (for iRMX 86 users) or the appropriate device name when calling DQ$ATTACH$FILE or DQ$CREATE$FILE (for iRMX 88 users). Before the DUIBs for a unit can be changed, however, the unit must be detached.

Figure 2-1 illustrates this concept. It shows six DUIBs, two for each of three units of one device. The main difference within each pair of DUIBs in this figure is the device granularity parameter, which is either 128 or 512. With this setup, a user can attach any unit of this device with one of two device granularities. In Figure 2-1, units 0 and 1 are attached with a granularity of 128 and unit 2 with a granularity of 512. To change this, the user can detach the device and attach it again using the other DUIB name.

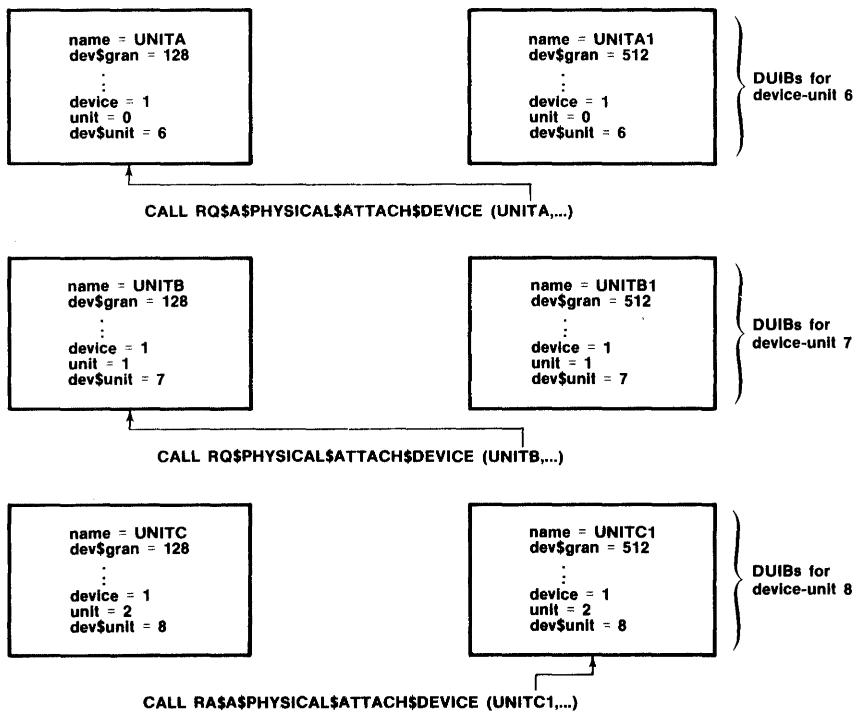

Figure 2-1. Attaching Devices

Creating DUIBs

Before the system starts running, you must provide all of the DUIBs that your system will ever need; you cannot create additional ones at run time. If you are an iRMX 88 user, you define your DUIBs during interactive configuration. If, on the other hand, you are an iRMX 86 user, place the DUIBs in the I/O System configuration file as a part of the I/O System configuration process. The iRMX 86 CONFIGURATION GUIDE describes this procedure. Observe the following guidelines when creating DUIBs:

- Specify a unique name for every DUIB, even those that describe the same device-unit.
- Create at least one DUIB for every device-unit in the hardware configuration. Since the DUIB contains the addresses of the device driver routines, this guarantees that no device-unit is left without a device driver to handle its I/O.

- Make sure to specify the same device driver procedures in all of the DUIBs associated with a particular device. There is only one set of device driver routines for a given device, and each DUIB for that device must specify this unique set of routines.

- If you are using a common or random access device driver, you must create a device information table for each device. If you are using a random access device driver, you must create a unit information table for each unit. See Chapter 4 for specifications of these tables. Place pointers to these tables in the device$info$p and unit$info$p fields of the appropriate DUIBs. If you are using custom device drivers and they require these or similar tables, you must create them, as well.

I/O REQUEST/RESULT SEGMENT (IORS)

An I/O request/result segment (IORS) is a data structure that the I/O System creates when a user requests an I/O operation. It contains information about the request and about the unit on which the operation is to be performed. The I/O System passes the IORS to the appropriate device driver which then processes the request. When the device driver performs the operation indicated in the IORS, it must modify the IORS to indicate what it has done and send the IORS back to the response mailbox (exchange) indicated in the IORS.

The IORS is the only mechanism that the I/O System uses to transmit requests to device drivers. Its structure is always the same. Every device driver must be aware of this structure and must update the information in the IORS after performing the requested function. The IORS is structured as follows:

```
DECLARE
    IORS                STRUCTURE(
          STATUS            WORD,
          UNIT$STATUS       WORD,
          ACTUAL            WORD,
          ACTUAL$FILL       WORD,
          DEVICE            WORD,
          UNIT              BYTE,
          FUNCT             BYTE,
          SUBFUNCT          WORD,
          LOW$DEV$LOC       WORD,
          HIGH$DEV$LOC      WORD,
          BUFF$P            POINTER,
          COUNT             WORD,
          COUNT$FILL        WORD,
          AUX$P             POINTER,
          LINK$FOR          POINTER,
          LINK$BACK         POINTER,
          RESP$MBOX         WORD,
          DONE              BYTE,
          FILL              BYTE,
          CANCEL$ID         WORD);
```

where:

STATUS	WORD in which the device driver must place the condition code for the I/O operation. The E$OK condition code indicates successful completion of the operation. For a complete list of possible condition codes, see either the iRMX 86 NUCLEUS REFERENCE MANUAL, the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL, and the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL, or the iRMX 88 REFERENCE MANUAL.
UNIT$STATUS	WORD in which the device driver must place additional status information if the status parameter was set to indicate the E$IO condition. The unit status codes and their descriptions are as follows:

code	mnemonic	description
0	IO$UNCLASS	Unclassified error
1	IO$SOFT	Soft error; a retry is possible
2	IO$HARD	Hard error; a retry is impossible
3	IO$OPRINT	Operator intervention is required.
4	IO$WRPROT	Write-protected volume

The I/O System reserves values 0 through 15 (the rightmost four bits) of this field for unit status codes. The high 12 bits of this field can be used for any other purpose that you wish. For example, the iSBC 204 driver places the result byte in the high eight bits of this field. Refer to the iSBC 204 FLEXIBLE DISKETTE CONTROLLER HARDWARE REFERENCE MANUAL for further information on the result byte.

ACTUAL	WORD which the device driver must update on the completion of an I/O operation to indicate the number of bytes of data actually transferred.
ACTUAL$FILL	Reserved WORD.
DEVICE	WORD in which the I/O System places the number of the device for which this request is intended.
UNIT	BYTE in which the I/O System places the number of the unit for which this request is intended.
FUNCT	BYTE in which the I/O System places the function code for the operation to be performed. Possible function codes are:

function	code
F$READ	0
F$WRITE	1
F$SEEK	2
F$SPECIAL	3
F$ATTACH$DEV	4
F$DETACH$DEV	5
F$OPEN	6
F$CLOSE	7

SUBFUNCT — WORD in which the I/O System places the actual function code of the operation, when the F$SPECIAL function code was placed in the FUNCT field. The value in this field depends on the device driver. The random access device driver currently supports the following two special functions:

function	code
FORMAT/QUERY	0
SATISY	1
NOTIFY	2

To maintain compatibility with random access device drivers and to allow for future expansion, other drivers should avoid using these codes, and 3 through 10 as well, for other functions.

LOWDEVLOC
HIGHDEVLOC — WORD pair that forms a 32-bit field in which the I/O System places the absolute byte location on the I/O device where the operation is to be performed. For example, for the F$WRITE operation, this is the address on the device where writing begins. If a random access device driver is used and the track$size field in the unit's unit information table contains a value greater than zero, this field contains the track number (in HIGHDEVLOC) and sector number (in LOWDEVLOC). If track$size contains zero, this field contains a 32-bit sector number.

BUFF$P — POINTER which the I/O System sets to indicate the internal buffer where data is read from or written to.

COUNT — WORD which the I/O System sets to indicate the number of bytes to transfer.

COUNT$FILL — Reserved WORD.

AUX$P — POINTER which the I/O System sets to indicate the location of auxiliary data. This data is used when the request calls the F$SPECIAL function, in order to pass a variety of kinds of special function data. For example, to format a track on a hard disk, set FUNCT equal to F$SPECIAL, set SUBFUNCT equal to 0, and set AUX$P to point to a structure of the form:

```
DECLARE
    FORMAT$TRACK     STRUCTURE(
        TRACK$NUMBER   WORD,
        INTERLEAVE     WORD,
        TRACK$OFFSET   WORD,
        FILL$CHAR      WORD);
```

The other Intel-defined F$SPECIAL options do not have predefined structures.

LINK$FOR — POINTER that the device driver can use to implement a request queue. Random access and common drivers use this field to point to the location of the next IORS in the queue.

LINK$BACK — POINTER that the device driver can use to implement a request queue. Random access and common drivers use this field to point to the location of the previous IORS in the queue.

RESP$MBOX — WORD that the I/O System fills with either an iRMX 86 token for the response mailbox or the address of an iRMX 88 exchange. Upon completion of the I/O request, the device driver must send the IORS to this response mailbox or exchange.

DONE — BYTE that the device driver can set to TRUE (OFFH) or FALSE (OOH) to indicate whether or not the entire request has been completed. Random access and common drivers use this byte in this fashion.

FILL — Reserved BYTE.

CANCEL$ID — WORD used to identify queued I/O requests that are to be removed from the queue by the CANCEL$IO procedure.

DEVICE INTERFACES

One or more of the routines in every device driver must actually send commands to the device itself in order to carry out I/O requests. The steps that a procedure of this sort must go through vary considerably, depending on the type of I/O device. Procedures supplied with the I/O System to manipulate devices such as the iSBC 204 and iSBC 206 devices use the PL/M-86 builtins INPUT and OUTPUT to transmit to and receive from I/O ports. Other devices may require different methods. The I/O System places no restrictions on the method of communicating with devices. Use the method that the device requires.

CHAPTER 3. CATEGORIES AND PROPERTIES OF DEVICES AND DRIVERS

There are three types of device drivers in the iRMX 86 and iRMX 88 environments: common, random access, and custom. This chapter defines the distinctions between the types of drivers and discusses the characteristics and data structures pertaining to common and random access device drivers. Chapters 5 and 6 are devoted to explaining how to write the various types of device drivers.

CATEGORIES OF DEVICES

Because the I/O Systems provide procedures that constitute the bulk of any common or random access device driver, you should consider the possibility that your device is a common or random access device. If your device falls in either of these categories, you can avoid most of the work of writing a device driver by using the supplied procedures. The following sections define the three types of devices.

COMMON DEVICES

Common devices are relatively simple devices, such as line printers, that conform to the following conditions:

- A first-in/first-out mechanism for queuing requests is sufficient for accessing these devices.

- Only one interrupt level is needed to service a device.

- Data either read or written by these devices does not need to be broken up into blocks.

If you have devices that fit into this category, you can save the effort of creating an entire device driver by using the common driver routines supplied by the I/O System. Chapter 5 of this manual describes the procedures that you must write to complete the balance of a common device driver.

RANDOM ACCESS DEVICES

A random access device is a device, such as a disk drive, in which data can be read from or written to any address of the device. The support routines provided by the I/O System for random access assume the following conditions:

- A first-in/first-out mechanism for queuing requests is sufficient for accessing these devices.

- Only one interrupt level is needed to service the device.

- I/O requests must be broken up into blocks of a specific length.

- The device supports random access seek operations.

If you have devices that fit into the random access category, you can take advantage of the random access support routines provided by the I/O System. Chapter 5 of this manual describes the procedures that you must write to complete the balance of a random access device driver.

CUSTOM DEVICES

If your device fits neither the common nor the random access category, you must write the entire driver for the device. The requirements of a custom device driver are defined in Chapter 6.

I/O SYSTEM-SUPPLIED ROUTINES FOR COMMON AND RANDOM ACCESS DEVICE DRIVERS

The I/O System supplies the common and random access routines that the I/O System calls when processing I/O requests. Flow charts for these procedures can be found in Appendix A; their names and functions are as follows:

Common Routine	Random Access Routine	Function
INIT$IO	RAD$INIT$IO	Creates the resources needed by the remainder of the driver routines, creates an interrupt task, and calls a user-supplied routine that initializes the device itself.
FINISH$IO	RAD$FINISH$IO	Deletes the resources used by the other driver routines, deletes the interrupt task, and calls a user-supplied procedure that performs final processing on the device itself.
QUEUE$IO	RAD$QUEUE$IO	Places I/O requests (IORSs) on the queue of requests.
CANCEL$IO	RAD$CANCEL$IO	Removes one or more requests from the request queue, possibly stopping the processing of a request that has already been started.

In addition to these routines, the I/O Systems supply an interrupt handler (interrupt service routine) and either INTERRUPT$TASK or RAD$INTERRUPT$TASK, which respond to all interrupts generated by the units of a device, process those interrupts, and start the device working on the next I/O request on the queue. This interrupt task is the one that the INIT$IO or RAD$INIT$IO procedure creates.

After a device finishes processing a request, it sends an interrupt to the processor. As a consequence, the processor calls the interrupt handler. This handler either processes the interrupt itself or invokes an interrupt task to process the interrupt. Since an interrupt handler is limited in the types of system calls that it can make and the number of interrupts that can be enabled while it is processing, an interrupt task usually services the interrupt. The interrupt task feeds the results of the interrupt back to the I/O System (data from a read operation, status from other types of operations). The interrupt task then gets the next I/O request from the queue and starts the device processing this request. This cycle continues until the device is detached.

Figure 3-1 shows the interaction between an interrupt task, an I/O device, an I/O request queue, and a Queue I/O device driver procedure. The interrupt task in this figure is in a continual cycle of waiting for an interrupt, processing it, getting the next I/O request, and starting up the device again. While this is going on, the Queue I/O procedure runs in parallel, putting additional I/O requests on the queue.

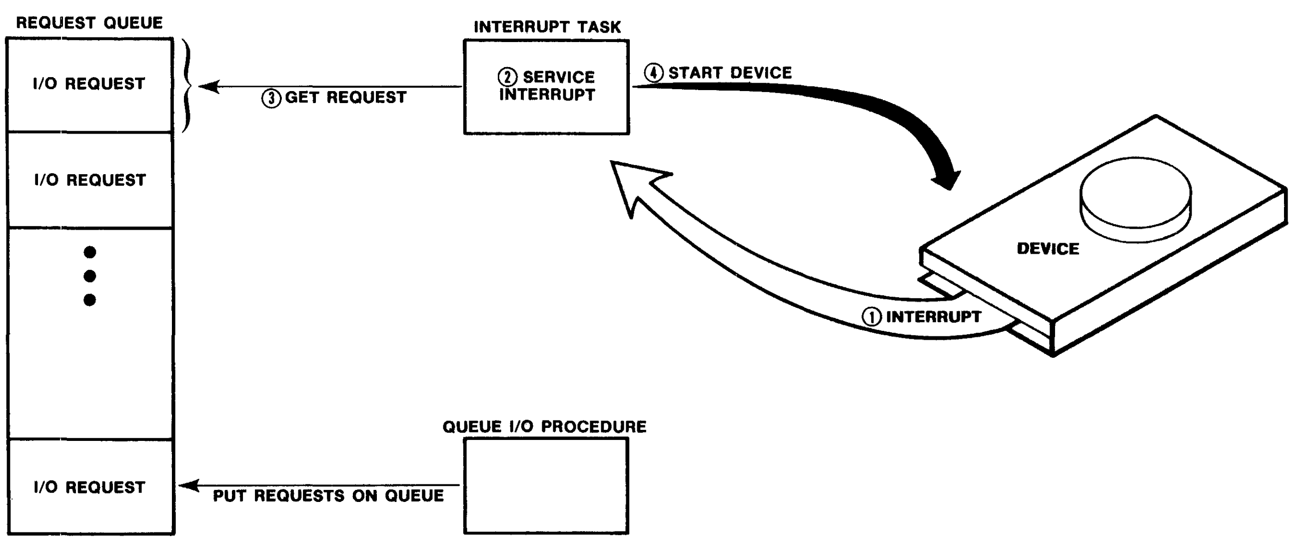

Figure 3-1. Interrupt Task Interaction

I/O SYSTEM ALGORITHM FOR CALLING THE DEVICE DRIVER PROCEDURES

The I/O System calls each of the four device driver procedures in response to specific conditions. Figure 3-2 is a flow chart that illustrates the conditions under which three of the four procedures are called. The following numbered paragraphs discuss the portions of Figure 3-2 labelled with corresponding circled numbers.

1. In order to start I/O processing, an application task must make an I/O request. This can be done by making any of a variety of system calls. However, if you are an iRMX 86 user, the first I/O request to each device-unit must be an RQAPHYSICAL$ATTACH$DEVICE system call, and if you are an iRMX 88 user, the first request to each device-unit must be either a DQ$ATTACH or a DQ$CREATE system call.

2. If the request results from an RQAPHYSICAL$ATTACH$DEVICE, a DQ$ATTACH, or a DQ$CREATE system call, the I/O System checks to see if any other units of the device are currently attached. If no other units of the device are currently attached, the I/O System realizes that the device has not been initialized and calls the Initialize I/O procedure first, before queueing the request.

3. Whether or not the I/O System called the Initialize I/O procedure, it calls the Queue I/O procedure to queue the request for execution.

4. If the request just queued resulted from an RQAPHYSICAL$DETACH$DEVICE system call, the I/O System checks to see if any other units of the device are currently attached. If no other units of the device are attached, the I/O System calls the Finish I/O procedure to do any final processing on the device and clean up resources used by the device driver routines.

The iRMX 86 I/O System calls the fourth device driver procedure, the Cancel I/O procedure, under the following conditions:

- If the user makes an RQAPHYSICAL$DETACH$DEVICE system call specifying the hard detach option, in order to forcibly detach the connection objects associated with a device-unit. The iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL describes the hard detach option.

- If the job containing the task which made a request is deleted.

The iRMX 88 I/O System does not call the Cancel I/O procedure.

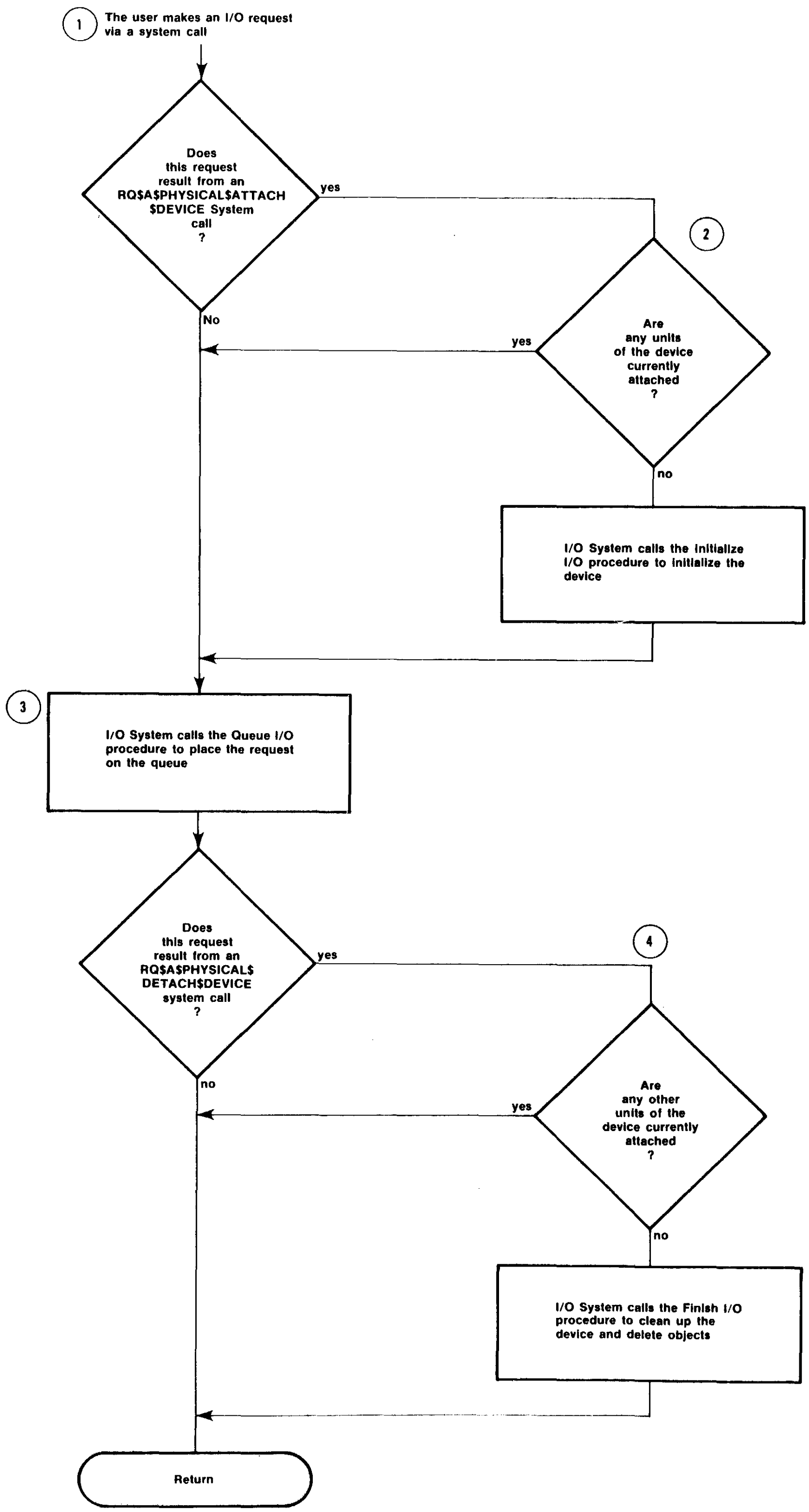

Figure 3-2. How the I/O System Calls the Device Driver Procedures

REQUIRED DATA STRUCTURES

In order for the I/O System-supplied routines to be able to call the user-supplied routines, you must include the addresses of these user-supplied routines, along with other information, in a device information table. In addition, processing I/O requests through a random access driver requires a unit information table. Each device-unit information block (DUIB) contains one pointer field for a device information table and another for a unit information table.

DUIBs which correspond to units of the same device should point to the same device information table, but they can point to different unit information tables, if the units have different characteristics. Figure 3-3 illustrates this.

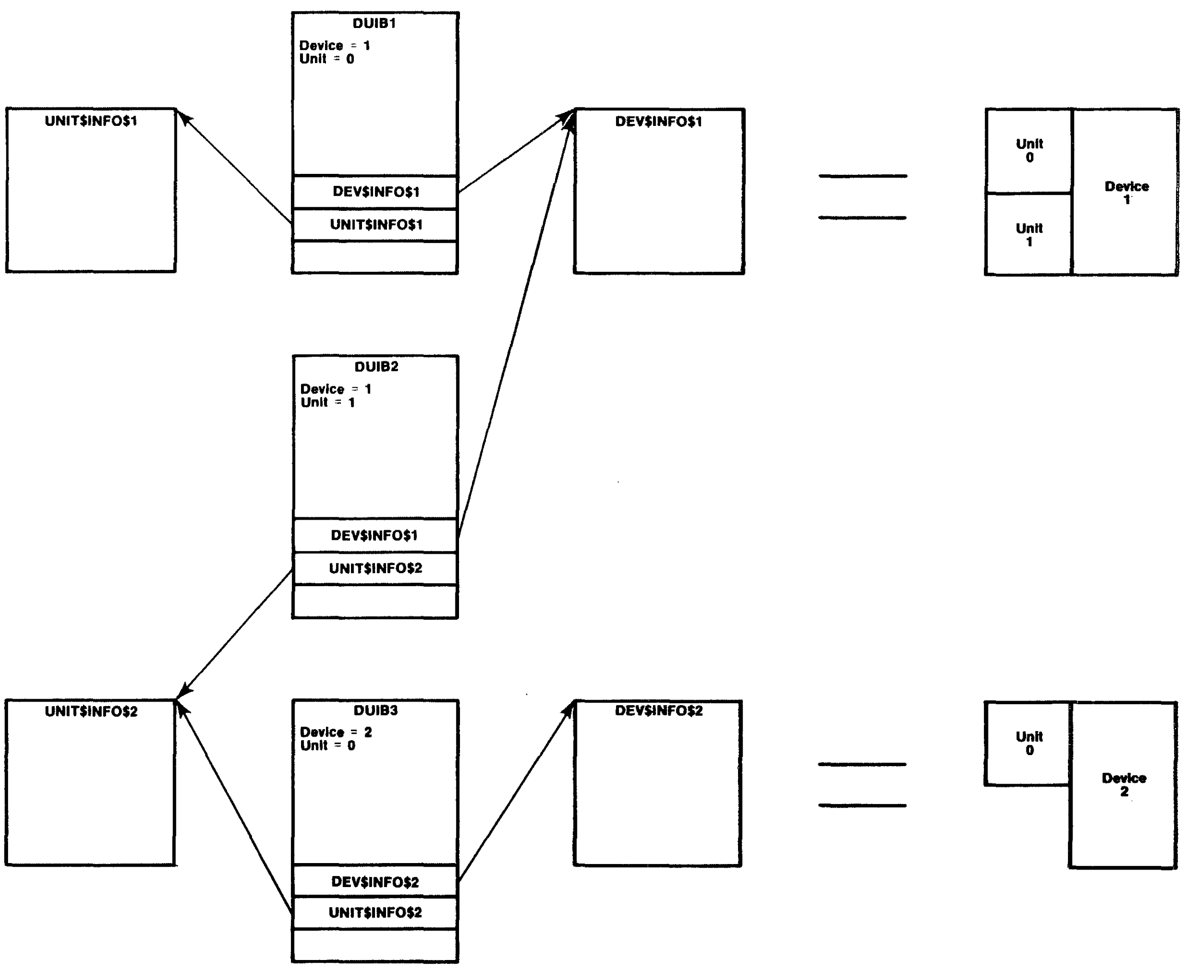

Figure 3-3. DUIBs, Device and Unit Information Tables

DEVICE INFORMATION TABLE

Common and random access device information tables contain the same fields in the same order, but the tables have different names. Common drivers refer to the device information table as COMMON$DEVICE$INFO, while random access drivers refer to RAD$DEVICE$INFO. For brevity, we show only the declaration of COMMON$DEVICE$INFO.

```
DECLARE
     COMMON$DEVICE$INFO     STRUCTURE(
          LEVEL                WORD,
          PRIORITY             BYTE,
          STACK$SIZE           WORD,
          DATA$SIZE            WORD,
          NUM$UNITS            WORD,
          DEVICE$INIT          WORD,
          DEVICE$FINISH        WORD,
          DEVICE$START         WORD,
          DEVICE$STOP          WORD,
          DEVICE$INTERRUPT     WORD);
```

where:

LEVEL — WORD specifying an encoded interrupt level at which the device will interrupt. The interrupt task uses this value in order to associate itself with the correct interrupt level. The values for this field are encoded as follows:

Bits	Value
15-7	0
6-4	First digit of the interrupt level (0-7)
3	If one, the level is a master level and bits 6-4 specify the entire level number. If zero, the level is a slave level and bits 2-0 specify the second digit.
2-0	Second digit of the interrupt level (0-7), if bit 3 is zero.

NOTE

In iRMX 88 systems, only master levels are available.

PRIORITY	BYTE specifying the initial priority of the interrupt task. The actual priority of an iRMX 86 interrupt task might change because the iRMX 86 Nucleus adjusts an interrupt task's priority according to the interrupt level that it services. Refer to the iRMX 86 NUCLEUS REFERENCE MANUAL for further information about this relationship between interrupt task priorities and interrupt levels.
STACK$SIZE	WORD specifying the size, in bytes, of the stack for the user-written device interrupt procedure (and procedures that it calls). This number should not include stack requirements for the I/O System-supplied procedures. They add their requirements to this figure.
DATA$SIZE	WORD specifying the size, in bytes, of the user portion of the device's data storage area. This figure should not include the amount needed by the I/O System-supplied procedures; rather, it should include only that amount needed by the user-written routines. This then is the size of the read or write buffers plus any flags that the user-written routines need.
NUM$UNITS	WORD specifying the number of units supported by the driver. Units are assumed to be numbered consecutively, starting with zero.
DEVICE$INIT	WORD specifying the start address of a user-written device initialization procedure. The format of this procedure, which is called by INIT$IO, is described in Chapter 5.
DEVICE$FINISH	WORD specifying the start address of a user-written device finish procedure. The format of this procedure, which is called by FINISH$IO, is described in Chapter 5.
DEVICE$START	WORD specifying the start address of a user-written device start procedure. The format of this procedure, which is called by QUEUE$IO and INTERRUPT$TASK, is described in Chapter 5.
DEVICE$STOP	WORD specifying the start address of a user-written device stop procedure. The format of this procedure, which is called by CANCEL$IO, is described in Chapter 5.
DEVICE$INTERRUPT	WORD specifying the start address of a user-written device interrupt procedure. The format of this procedure, which is called by INTERRUPT$TASK, is described in Chapter 5.

Depending on the requirements of your device, you can append additional information to the COMMON$DEVICE$INFO or RAD$DEVICE$INFO structure. For example, most devices require that the I/O port address be appended to this structure, in order that the user-written procedures have access to the device.

You must create device information tables as a part of the I/O System configuration process. The iRMX 86 CONFIGURATION GUIDE describes this procedure for iRMX 86 users. You configure iRMX 88 applications interactively with the Interactive Configuration Utility.

UNIT INFORMATION TABLE

If you have random access device drivers in your system, you must create a unit information table for each different type of unit in your system. Each random access device-unit's DUIB must point to one unit information table, although multiple DUIBs can point to the same unit information table. The structure of the unit information table is as follows:

```
DECLARE
    RAD$UNIT$INFO     STRUCTURE(
        TRACK$SIZE        WORD,
        MAX$RETRY         WORD,
        RESERVED          WORD);
```

where:

TRACK$SIZE — WORD specifying the size, in bytes, of a single track of a volume on the unit. If the device controller supports reading and writing across track boundaries, place a zero in this field. If you specify a zero for this field, the I/O System-supplied procedures place a 32-bit sector number in the HIGHDEVLOC and LOWDEVLOC fields of the IORS. If you specify a nonzero value for this field, the I/O System-supplied procedures guarantee that read and write requests do not cross track boundaries. They do this by placing the sector number in the low order word of the LOWDEVLOC field of the IORS and the track number in the HIGHDEVLOC field before calling a user-written device start procedure. Instructions for writing a device start procedure are contained in Chapter 5.

MAX$RETRY — WORD specifying the maximum number of times an I/O request should be tried if an error occurs. A value of nine is recommended for this field. When this field contains a nonzero value, the I/O System-supplied procedures guarantee that read or write requests are retried if the user-supplied device start or device interrupt procedures return an IO$SOFT condition in the IORS.UNIT$STATUS field. (The IORS.UNIT$STATUS field is described in the "IORS Structure" section of Chapter 2.)

RESERVED — Reserved WORD.

RELATIONSHIPS BETWEEN I/O PROCEDURES AND I/O DATA STRUCTURES

This section brings together several of the procedures and data structures that have been described so far in this manual. Figure 3-4 shows the many relationships that exist among these entities, with solid arrows indicating procedure calls and dotted arrows indicating pointers. Note that the I/O System contains the address of each DUIB, which in turn contains the addresses of the procedures that the I/O System calls in order to perform I/O on the associated device-unit. The DUIB also has the address of the device information table and, if the device is a random access device, the unit information table. The device information table, in turn, contains the addresses of the procedures that are called by the procedures that the I/O System calls. It is through these links that the appropriate calls are made in the servicing of an I/O request for a particular device-unit.

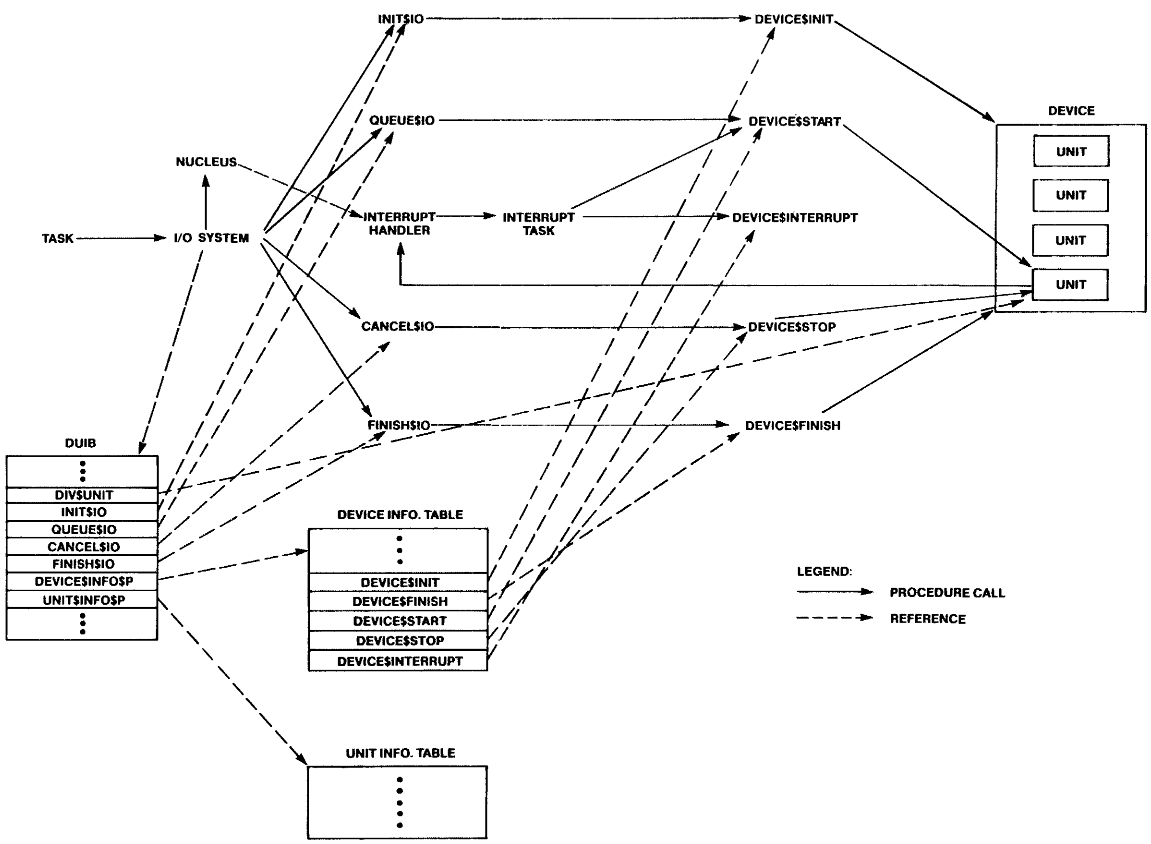

Figure 3-4. Relationships Between I/O Procedures and I/O Data Structures

WRITING DRIVERS FOR USE WITH BOTH iRMX 86- AND iRMX 88-BASED SYSTEMS

A common or random access device driver that makes no system calls will be compatible with both the iRMX 86 and iRMX 88 I/O Systems. Consequently, such a device driver can be "ported" between applications based on the two iRMX systems.

DEVICE DATA STORAGE AREA

The common and random access device drivers are set up so that all data which is local to a device is maintained in an area of memory. The Initialize I/O procedure creates this device data storage area and the other procedures of the driver access and update information in it as needed. Two purposes are served by storing the device-local data in a central area.

First, all device driver procedures which service individual units of the device can access and update the same data. The Initialize I/O procedure passes the address of the area back to the I/O System, which in turn gives the address to the other procedures of the driver. They can then place information relevant to the device as a whole into the area. The identity of the first IORS on the request queue is maintained in this area, as well as the attachment status of the individual units and a means of accessing the interrupt task.

Second, several devices of the same type can share the same device driver code and still maintain separate device data areas. For example, suppose two iSBC 204 devices use the same device driver code. The same Initialize I/O procedure is called for each device, and each time it is called it obtains memory for the device data. However, the memory areas that it creates are different. Only the incarnations of the routines that service units of a particular device are able to access the device data area for that device.

Although the common and random access device drivers already provide this mechanism, you may want to include a device data storage area in any custom driver that you write.

CHAPTER 4. I/O REQUESTS

This chapter contains two kinds of information that writers of device drivers will find useful. Presented first are summaries of the actions that the I/O System takes in response to the various kinds of I/O requests that application tasks can make. Next are three tables --- one for each type of device driver --- that show which DUIB and IORS fields device drivers should be concerned with.

I/O SYSTEM RESPONSES TO I/O REQUESTS

This section shows which device driver procedures the I/O System calls when it processes each of the eight kinds of I/O requests. When there are multiple calls, the order of the calls is significant.

ATTACH DEVICE REQUESTS

When the I/O System receives the first attach device request for a device, it makes the following calls to device driver procedures in the following order:

The Call	The Effects of the Call
Initialize I/O	The driver resets the device as a whole and creates the device data storage area and interrupt tasks.
Queue I/O, with the FUNCT field of the IORS set to F$ATTACH (=4)	The driver resets the selected unit.

When the I/O System receives an attach device request that is not the first for the device, it makes the following call:

The Call	The Effects of the Call
Queue I/O, with the FUNCT field of the IORS set to F$ATTACH (=4)	The driver resets the selected unit.

DETACH DEVICE REQUESTS

When the I/O System receives a detach device request, and there is more than one unit of the device attached, it makes the following call:

The Call	The Effects of the Call
Queue I/O, with the FUNCT field of the IORS set to F$DETACH (=5)	The driver performs cleanup operations for the selected unit, if necessary.

When the I/O System receives a detach device request, and there is only one attached unit on the device, it makes the following calls to device driver procedures in the following order:

The Call	The Effects of the Call
Queue I/O, with the FUNCT field of the IORS set to F$DETACH (=5)	The driver performs cleanup operations for the selected unit, if necessary.
Finish I/O	The driver performs cleanup operations for the device as a whole, if necessary.

READ, WRITE, OPEN, CLOSE, SEEK, AND SPECIAL REQUESTS

When the I/O System receives a read, write, open, close, seek, or special request, it makes the following call to a device driver procedure:

The Call	The Effects of the Call
Queue I/O, with the FUNCT field of the IORS set to F$READ (=0), F$WRITE (=1), F$OPEN (=6), F$CLOSE (=7), F$SEEK (=2), or F$SPECIAL (=3), depending on the type of the I/O request.	The driver performs the requested operation. (F$OPEN and F$CLOSE usually require no processing.)

CANCEL REQUESTS

When a connection is deleted while I/O might be in progress, such as when an iRMX 86 job is deleted, the I/O System makes the following calls to device driver procedures in the following order:

The Call	The Effects of the Call
Cancel I/O.	The driver removes from the request queue all requests that contain the same Cancel ID value as that in the current request, and stops processing if necessary.
Queue I/O, with the FUNCT field of the IORS set to F$CLOSE (=7)	When this request reaches the front of the queue, it is simply returned to the indicated response mailbox (exchange).

DUIB AND IORS FIELDS USED BY DEVICE DRIVERS

The following tables indicate, for each type of device driver, the fields of DUIBs and IORSs with which user-written portions of device drivers need to be concerned.

Table 4-1. DUIB and IORS Fields Used by Common Device Drivers

	Attach Device	Detach Device	Open	Close	Read	Write	Seek	Special
DUIB								
Name								
File$drivers								
Functs								
Flags	m	m	m	m	m	m	m	m
Dev$gran	m	m	m	m	m	m	m	m
Dev$size	m	m	m	m	m	m	m	m
Device								
Unit	m	m	m	m	m	m	m	m
Dev$unit								
Init$io								
Finish$io								
Queue$io								
Cancel$io								
Device$info$p	m	m	m	m	m	m	m	m
Unit$info$p	m	m	m	m	m	m	m	m
Update$timeout								
Num$buffers								
Priority								
IORS								
Status	w	w	w	w	w	w	w	w
Unit$status	w	w	w	w	w	w	w	w
Actual					w	w		
Actual$fill								
Device								
Unit	m	m	m	m	m	m	m	m
Funct	r	r	r	r	r	r	r	r
Subfunct								r
Lowdevloc					m	m	m	
Highdevloc					m	m	m	
Buff$p					r	r		
Count					r	r		
Count$fill								
Aux$p								m
Link$for								
Link$back								
Resp$mbox								
Done	w	w	w	w	w	w	w	w
Fill								
Cancel$id								

r --- is read by the device driver
w --- is written by the device driver
m --- might be read by some device drivers

Table 4-2. DUIB and IORS Fields Used by Random Access Device Drivers

	Attach Device	Detach Device	Open	Close	Read	Write	Seek	Special
DUIB								
Name								
File$drivers								
Functs								
Flags	m	m	m	m	m	m	m	m
Dev$gran	m	m	m	m	m	m	m	m
Dev$size	m	m	m	m	m	m	m	m
Device								
Unit	m	m	m	m	m	m	m	m
Dev$unit								
Init$io								
Finish$io								
Queue$io								
Cancel$io								
Device$info$p	m	m	m	m	m	m	m	m
Unit$info$p	m	m	m	m	m	m	m	m
Update$timeout								
Num$buffers								
Priority								
IORS								
Status	w	w	w	w	w	w	w	w
Unit$status	w	w	w	w	w	w	w	w
Actual					w	w		
Actual$fill								
Device								
Unit	m	m	m	m	m	m	m	m
Funct	r	r	r	r	r	r	r	r
Subfunct								r
Lowdevloc					r	r	r	
Highdevloc					r	r	r	
Buff$p					r	r		
Count					r	r		
Count$fill								
Aux$p								m
Link$for								
Link$back								
Resp$mbox								
Done	w	w	w	w	w	w	w	w
Fill								
Cancel$id								

r --- is read by the device driver
w --- is written by the device driver
m --- might be read by some device drivers

Table 4-3. DUIB and IORS Fields Used by Custom Device Drivers

	Attach Device	Detach Device	Open	Close	Read	Write	Seek	Special
DUIB								
Name								
File$drivers								
Functs								
Flags	m	m	m	m	m	m	m	m
Dev$gran	m	m	m	m	m	m	m	m
Dev$size	m	m	m	m	m	m	m	m
Device								
Unit	m	m	m	m	m	m	m	m
Dev$unit								
Init$io								
Finish$io								
Queue$io								
Cancel$io								
Device$info$p	m	m	m	m	m	m	m	m
Unit$info$p	m	m	m	m	m	m	m	m
Update$timeout								
Num$buffers								
Priority								
IORS								
Status	w	w	w	w	w	w	w	w
Unit$status	w	w	w	w	w	w	w	w
Actual					w	w		
Actual$fill								
Device								
Unit	m	m	m	m	m	m	m	m
Funct	r	r	r	r	r	r	r	r
Subfunct								
Lowdevloc					m	m	m	
Highdevloc					m	m	m	
Buff$p					r	r		
Count					r	r		
Count$fill								
Aux$p								m
Link$for	a	a	a	a	a	a	a	a
Link$back	a	a	a	a	a	a	a	a
Resp$mbox	r	r	r	r	r	r	r	r
Done	a	a	a	a	a	a	a	a
Fill	a	a	a	a	a	a	a	a
Cancel$id				m				

r --- is read by the device driver
w --- is written by the device driver
m --- might be read by some device drivers
a --- is available for any purpose suiting the needs of the device driver

CHAPTER 5. WRITING COMMON OR RANDOM ACCESS DEVICE DRIVERS

This chapter contains the calling sequences for the procedures that you must provide when writing a common or random access device driver. Where possible, descriptions of the duties of these procedures accompany the calling sequences.

The I/O System-supplied procedures are referred to in this chapter, for brevity, as if the chapter were written only for writers of common device drivers. For example, "INIT$IO" is shorthand for "INIT$IO or RAD$INIT$IO".

GENERAL INFORMATION

The routines that are provided by the I/O System and that the I/O System calls comprise the bulk of a common or random access a device driver. These routines, in turn, make calls to device-dependent routines that you must supply. These device-dependent routines are described here briefly and then are presented in detail:

A device initialization procedure. This procedure must perform any initialization functions necessary to get the device ready to process I/O. INIT$IO calls this procedure.

A device finish procedure. This procedure must perform any necessary final processing on the device so that the device can be detached. FINISH$IO calls this procedure.

A device start procedure. This procedure must start the device processing any possible I/O function. QUEUE$IO and INTERRUPT$TASK (the I/O System-supplied interrupt task) call this procedure.

A device stop procedure. This procedure must stop the device from processing the current I/O function, if that function could take an indefinite amount of time. CANCEL$IO calls this procedure.

A device interrupt procedure This procedure must do all of the device-dependent processing that results from the device sending an interrupt. INTERRUPT$TASK calls this procedure.

DEVICE INITIALIZATION PROCEDURE

The INIT$IO procedure calls the user-written device initialization procedure in order to initialize the device. The format of the call to the user-written device initialization procedure is as follows:

```
CALL device$init(duib$p, ddata$p, status$p);
```

where:

device$init	Name of the device initialization procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the device information table.
duib$p	POINTER to the DUIB of the device-unit being attached. From this DUIB, the device initialization procedure can obtain the device information table, where information such as the I/O port address is stored.
ddata$p	POINTER to the user portion of the device's data storage area. You must specify the size of this portion in the device information table for this device. The device initialization procedure can use this data area for whatever purposes it chooses. Possible uses for this data area include local flags and buffer areas.
status$p	POINTER to a WORD in which the device initialization procedure must return the status of the initialization operation. It should return the E$OK condition code if the initialization is successful; otherwise it should return the appropriate exceptional condition code. If initialization does not complete successfully, the device initialization procedure must ensure that any data areas it initializes are reset.

If you have a device that does not need to be initialized before it can be used, you can use the default device initialization procedure supplied by the I/O System. The name of this procedure is DEFAULT$INIT. Specify this name in the device information table. DEFAULT$INIT does nothing but return the E$OK condition code.

DEVICE FINISH PROCEDURE

The FINISH$IO procedure calls the user-written device finish procedure in order to perform final processing on the device, after the last I/O request has been processed. The format of the call to the device finish procedure is as follows:

```
CALL device$finish(duib$p, ddata$p);
```

where:

device$finish	Name of the device finish procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the device information table.

duib$p	POINTER to the DUIB of the device-unit being detached. From this DUIB, the device finish procedure can obtain the device information table, where information such as the I/O port address is stored.
ddata$p	POINTER to the user portion of the device's data storage area. The device finish procedure should obtain, from this data area, identification of any resources other user-written procedures may have created, and delete these resources.

If you have a device that does not require any final processing, you can use the default device finish procedure supplied by the I/O System. The name of this procedure is DEFAULT$FINISH. Specify this name in the device information table. DEFAULT$FINISH merely returns to the caller with an E$OK condition code and is normally used when the default initialization procedure DEFAULT$INIT is used.

DEVICE START PROCEDURE

Both QUEUE$IO and INTERRUPT$TASK make calls to the device start procedure in order to start an I/O function. QUEUE$IO calls this procedure on receiving an I/O request when the request queue is empty. INTERRUPT$TASK calls the device start procedure after it finishes one I/O request if there are more I/O requests on the queue. The format of the call to the device start procedure is as follows:

```
CALL device$start(iors$p, duib$p, ddata$p);
```

where:

device$start	Name of the device start procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the device information table.
iors$p	POINTER to the IORS of the request. The device start procedure must access the IORS in order to obtain information such as the type of I/O function requested, the address on the device of the byte where I/O is to commence, and the buffer address.
duib$p	POINTER to the DUIB of the device-unit for which the I/O request is intended. The device start procedure can use the DUIB to access the device information table, where information such as the I/O port address is stored.
ddata$p	POINTER to the user portion of the device's data storage area. The device start procedure can use this data area to set flags or store data.

The device start procedure must do the following:

- It must be able to start the device processing any of the functions supported by the device and recognize that requests for nonsupported functions are error conditions.

- If it transfers any data, it must update the IORS.ACTUAL field to reflect the total number of bytes of data transferred (that is, if it transfers 128 bytes of data, it must put 128 in the IORS.ACTUAL field).

- If an error occurs when the device start procedure tries to start the device (such as on an F$WRITE request to a write-protected disk), the device start procedure must set the IORS.STATUS field to indicate an E$IO condition and the IORS.UNIT$STATUS field to a nonzero value. The lower four bits of the field should be set as indicated in the "IORS Structure" section of Chapter 2. The remaining bits of the field can be set to any value (for example, the iSBC 204 device driver returns the device's result byte in the remainder of this field). If the function completes without an error, the device start procedure must set the IORS.STATUS field to indicate an E$OK condition.

- If the device start procedure determines that the I/O request has been processed completely, either because of an error or because the request has been successfully completed, it must set the IORS.DONE field to TRUE. The I/O request will not always be completed; it may take several calls to the device interrupt procedure before a request is completed. However, if the request is finished and the device start procedure does not set the IORS.DONE field to TRUE, the device driver support routines will wait until the device sends an interrupt and the device interrupt procedure sets IORS.DONE to TRUE, before determining that the request is actually finished.

DEVICE STOP PROCEDURE

The CANCEL$IO procedure calls the user-written device stop procedure in order to stop the device from performing the current I/O function. The format of the call to the device stop procedure is as follows:

```
CALL device$stop(iors$p, duib$p, ddata$p);
```

where:

device$stop — Name of the device stop procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include this name in the device information table.

iors$p	POINTER to the IORS of the request. The device stop procedure needs this information to determine what type of function to stop.
duib$p	POINTER to the DUIB of the device-unit on which the I/O function is being performed.
ddata$p	POINTER to the user portion of the device's data storage area. The device stop procedure can use this area to store data, if necessary.

If you have a device which guarantees that all I/O requests will finish in an acceptable amount of time, you can omit writing a device stop procedure and use the default procedure supplied with the I/O System. The name of this procedure is DEFAULT$STOP. Specify this name in the device information table. DEFAULT$STOP simply returns to the caller.

DEVICE INTERRUPT PROCEDURE

INTERRUPT$TASK calls the user-written device interrupt procedure to process an interrupt that just occurred. Whereas the device start procedure is called to start the device performing an I/O function, the device interrupt procedure is called when the device finishes performing the function. The format of the call to the device interrupt procedure is as follows:

```
CALL device$interrupt(iors$p, duib$p, ddata$p);
```

where:

device$interrupt	Name of the device interrupt procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include this name in the device information table.
iors$p	POINTER to the IORS of the request being processed. The device interrupt procedure must update information in this IORS. A value of zero for this parameter indicates that there are no requests on the request queue and that the interrupt is extraneous.
duib$p	POINTER to the DUIB of the device-unit on which the I/O function was performed.
ddata$p	POINTER to the user portion of the device's data storage area. The device interrupt procedure can update flags in this data area or retrieve data sent by the device.

The device interrupt procedure must do the following:

- It must determine whether the interrupt resulted from the completion of an I/O function by the correct device-unit.
- If the correct device-unit did send the interrupt, the device interrupt procedure must determine whether the request is finished. If the request is finished, the device interrupt procedure must set the IORS.DONE field to TRUE.
- It must process the interrupt. This may involve setting flags in the user portion of the data storage area, tranferring data written by the device to a buffer, or some other operation.
- If an error has occurred, it must set the IORS.STATUS field to indicate an E$IO condition and the IORS.UNIT$STATUS field to a nonzero value. The lower four bits of the field should be set as indicated in the "IORS Structure" section of Chapter 2. The remaining bits of the field can be set to any value (for example, the iSBC 204 and 206 device drivers return the device's result byte in the remainder of this field). It must also set the IORS.DONE field to TRUE, indicating that the request is finished because of the error.
- If no error has occurred, it must set the IORS.STATUS field to indicate an E$OK condition.

CHAPTER 6. WRITING A CUSTOM DEVICE DRIVER

Custom device drivers are drivers that you create in their entirety because your device doesn't fit into either the common or random access device category, either because the device requires a priority-ordered queue, multiple interrupt levels, or because of some other reasons that you have determined. When you write a custom device driver, you must provide all of the features of the driver, including creating and deleting resources, implementing a request queue, and creating an interrupt handler. You can do this in any manner that you choose as long as you supply the following four procedures that the I/O System can call:

<u>An Initialize I/O Procedure</u>. This procedure must initialize the device and create any resources needed by the procedures in the driver.

<u>A Finish I/O Procedure</u>. This procedure must perform any final processing on the device and delete resources created by the remainder of the procedures in the driver.

<u>A Queue I/O Procedure</u>. This procedure must place the I/O requests on a queue of some sort, so that the device can process them when it becomes available.

<u>A Cancel I/O Procedure</u>. This procedure must cancel a previously queued I/O request.

In order for the I/O System to communicate with your device driver procedures, you must include the addresses of these four procedures in the DUIBs which correspond to the units of the device.

The next four sections describe the format of each of the I/O System calls to these four procedures. Your procedures must conform to these formats.

INITIALIZE I/O PROCEDURE

The iRMX 86 I/O System calls the Initialize I/O procedure when an application task makes an RQAPHYSICAL$ATTACH$DEVICE system call and no units of the device are currently attached. The iRMX 88 I/O System calls the Initialize I/O procedure when an application task attaches or creates a file on the device and no other files on the device are currently attached. In either case, the I/O System calls the Initialize I/O procedure before calling any other driver procedure.

The Initialize I/O procedure must perform any initial processing necessary for the device or the driver. If the device requires an interrupt task (or region or device data object, in the case of iRMX 86 drivers), the Initialize I/O procedure should create it (them).

The format of the call to the Initialize I/O procedure is as follows:

```
CALL init$io(duib$p, ddata$p, status$p);
```

where:

init$io — Name of the Initialize I/O procedure. You can use any name for this procedure as long as it does not conflict with other procedure names. You must, however, include its starting address in the DUIBs of all device-units that it services.

duib$p — POINTER to the DUIB of the device-unit for which the request is intended. The init$io procedure uses this DUIB to determine the characteristics of the unit.

ddata$p — POINTER to a WORD in which the init$io procedure can place the location of a data storage area, if the device driver needs such an area. If the device driver requires that a data area be associated with a device (to contain the head of the I/O queue, DUIB addresses, or status information), the init$io procedure should create this area and save its location via this pointer. If the driver does not need such a data area, the init$io procedure should return a zero via this pointer.

status$p — POINTER to a WORD in which the init$io procedure must place the status of the initialize operation. If the operation completes successfully, the init$io procedure must return the E$OK condition code. Otherwise it should return the appropriate exception code. If the init$io procedure does not return the E$OK condition code, it must delete any resources that it has created and leave all data fields with exactly the same information that they contained prior to the call to init$io.

FINISH I/O PROCEDURE

The iRMX 86 I/O System calls the Finish I/O procedure after an application task makes an RQAPHYSICAL$DETACH$DEVICE system call to detach the last unit of a device. The iRMX 88 I/O System calls the Finish I/O procedure when an application task detaches or deletes the last remaining file connection for the device.

The Finish I/O procedure performs any necessary final processing on the device. It must delete all resources created by other procedures in the device driver and must perform final processing on the device itself, if the device requires such processing.

The format of the call to the Finish I/O procedure is as follows:

```
CALL finish$io(duib$p, ddata$t);
```

where:

finish$io	Name of the Finish I/O procedure. You can specify any name for this procedure as long as it does not conflict with other procedure names. You must, however, include its starting address in the DUIBs of all device-units that it services.
duib$p	POINTER to the DUIB of a device-unit of the device being detached. The finish$io procedure needs this DUIB in order to determine the device on which to perform the final processing.
ddata$t	WORD containing the location of the data storage area originally created by the init$io procedure. The finish$io procedure must delete this resource and any others created by driver routines.

QUEUE I/O PROCEDURE

The I/O System calls the Queue I/O procedure to place an I/O request on a queue, so that it can be processed when the device is not busy. It is recommended that the Queue I/O procedure actually start the processing of the I/O request if the device is not busy. The format of the call to the Queue I/O procedure is as follows:

```
CALL queue$io(iors$t, duib$p, ddata$t);
```

where:

queue$io	Name of the Queue I/O procedure. You can use any name for this procedure as long as it does not conflict with other procedure names. You must, however, include its starting address in the DUIBs of all device-units that it services.
iors$t	WORD containing the location of an IORS. This IORS describes the request. When the request is processed, the driver (though not necessarily the queue$io procedure) must fill in the status fields and send the IORS to the response mailbox (exchange) indicated in the IORS. Chapter 2 describes the format of the IORS. It lists the

information that the I/O System supplies when it passes the IORS to the queue$io procedure and indicates the fields of the IORS that the device driver must fill in.

duib$p — POINTER to the DUIB of the device-unit for which the request is intended.

ddata$t — WORD containing the location of the data storage area originally created by the init$io procedure. The queue$io procedure can place any necessary information in this area in order to update the request queue or status fields.

CANCEL I/O PROCEDURE

The I/O System can call the Cancel I/O procedure in order to cancel one or more previously queued I/O requests. The iRMX 88 I/O System does not call Cancel I/O, but in the iRMX 86 environment Cancel I/O is called under either of the following two conditions:

- If the user makes an RQAPHYSICAL$DETACH$DEVICE system call and specifies the hard detach option (refer to the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL for a description of this call). This system call forcibly detaches all objects associated with a device-unit.

- If the job containing the task which made an I/O request is deleted. The I/O System calls the Cancel I/O procedure to remove any requests that tasks in the deleted job might have made.

If the device cannot guarantee that a request will be finished within a fixed amount of time (such as waiting for input from a terminal keyboard), the Cancel I/O procedure must actually stop the device from processing the request. If the device guarantees that all requests finish in an acceptable amount of time, the Cancel I/O procedure does not have to stop the device itself, but only removes requests from the queue.

The format of the call to the Cancel I/O procedure is as follows:

```
CALL cancel$io(cancel$id, duib$p, ddata$t);
```

where:

cancel$io — Name of the Cancel I/O procedure. You can use any name for this procedure as long as it doesn't conflict with other procedure names. You must, however, include its starting address in the DUIBs of all device-units that it services.

cancel$id — WORD containing the id value for the I/O requests that are are to be cancelled. Any pending requests with this value in the cancel$id field of their

IORS's must be removed from the queue of requests by the Cancel I/O procedure. Moreover, the I/O System places a CLOSE request with the same cancel$id value in the queue. The CLOSE request must not be processed until all other requests with that cancel$id value have been returned to the I/O System.

duib$p — POINTER to the DUIB of the device-unit for which the request cancellation is intended.

ddata$t — WORD containing the location of the data storage area originally created by the init$io procedure. This area may contain the request queue.

IMPLEMENTING A REQUEST QUEUE

Making I/O requests via system calls and the actual processing of these requests by I/O devices are asynchronous activities. When a device is processing one request, many more can be accumulating. Unless the device driver has a mechanism for placing I/O requests on a queue of some sort, these requests will become lost. The common and random access device drivers form this queue by creating a doubly linked list. The list is used by the QUEUE$IO and CANCEL$IO procedures, as well as by INTERRUPT$TASK.

Using this mechanism of the doubly linked list, common and random access device drivers implement a FIFO queue for I/O requests. If you are writing a custom device driver, you might want to take advantage of the LINK$FOR and LINK$BACK fields that are provided in the IORS and implement a scheme similar to the following for queuing I/O requests.

Each time a user makes an I/O request, the I/O System passes an IORS for this request to the device driver, in particular to the Queue I/O procedure of the device driver. The common and random access driver Queue I/O procedures make use of the LINK$FOR and LINK$BACK fields of the IORS to link this IORS together with IORSs for other requests that have not yet been processed.

This queue is set up in the following manner. The device driver routine that is actually sending data to the controller accesses the first IORS on the queue. The LINK$FOR field in this IORS points to the next IORS on the queue. The LINK$FOR field in the second IORS points to the third IORS on the queue, and so forth until, in the last IORS on the queue, the LINK$FOR field points back to the first IORS on the queue. The LINK$BACK fields operate in the same manner. The LINK$BACK field of the last IORS on the queue points to the previous IORS. The LINK$BACK field of the second to last IORS points to the third to last IORS on the queue, and so forth, until, in the first IORS on the queue, the LINK$BACK field points back to the last IORS in the queue. A queue of this sort is illustrated in Figure 6-1.

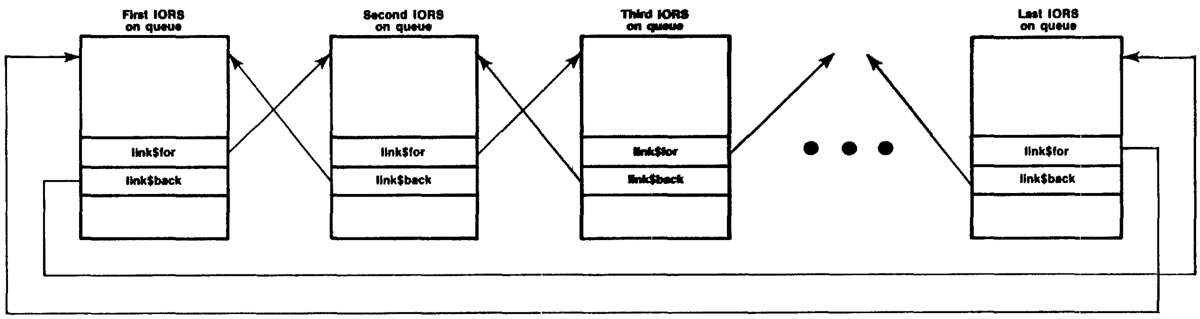

Figure 6-1. Request Queue

The device driver can add or remove requests from the queue by adjusting LINK$FOR and LINK$BACK pointers in the IORSs.

To handle the dual problems of locating the queue and ascertaining whether the queue is empty, you can use a variable such as head$queue. If the queue is empty, head$queue contains the value 0. Otherwise, head$queue contains the address of the first IORS in the queue.

CHAPTER 7. BINDING A DEVICE DRIVER TO THE I/O SYSTEM

You can write the modules for your device driver in either PL/M-86 or the MCS-86 Macro Assembly Language. However, you must adhere to the following guidelines:

- If you use PL/M-86, you must define your routines as reentrant, public procedures, and compile them using the ROM and COMPACT controls.

- If you use assembly language, your routines must follow the conditions and conventions used by the PL/M-86 COMPACT model of computation. In particular, your routines must function in the same manner as reentrant PL/M-86 procedures with the ROM and COMPACT controls set. The 8086/8087/8088 MACRO ASSEMBLER OPERATING INSTRUCTIONS FOR 8080/8085-BASED DEVELOPMENT SYSTEMS and the 8086/8087/8088 MACRO ASSEMBLER OPERATING INSTRUCTIONS FOR 8086-BASED DEVELOPMENT SYSTEMS describe these conditions and conventions.

If you are an iRMX 88 user, use the Interactive Configuration Utility to link your modules together and to the rest of the iRMX 88 executive.

The remainder of this chapter applies only to iRMX 86 users.

After you have created your device driver procedures and compiled or assembled them, you must link the object code to the I/O System. If you have written driver procedures for several types of devices, you might want to place all of these routines in a library and link this library to the I/O System. This allows you to maintain one file of driver routines and still link in only those routines that satisfy external references. The LIB86 command which allows you to create libraries of object modules is described in the 8086 FAMILY UTILITIES USER'S GUIDE FOR 8080/8085-BASED DEVELOPMENT SYSTEMS and the iAPX 86 FAMILY UTILITIES USER'S GUIDE FOR 8086-BASED DEVELOPMENT SYSTEMS.

The process of linking your driver procedures to the I/O System occurs at I/O System configuration time. The iRMX 86 CONFIGURATION GUIDE contains a description of this process. However, because the order in which you link the modules is important and because you must modify the Submit file IOS.CSD, this chapter contains a brief description of the required LINK86 command.

The command used to link the I/O System is as follows:

```
LINK86                  &
    :fx:ITABLE.OBJ,     &
    :fx:IDEVCF.OBJ,     &
    :fx:driver.obj,     &
    :fx:IOOPT1.LIB,     &
    :fx:IOS.LIB,        &
    :fx:RPIFC.LIB       &
TO  :fx:ios.lnk        (linker options)
```

where:

fx	The appropriate disk mnemonic, indicating where the file resides.
ITABLE.OBJ IDEVCF.OBJ	The assembled I/O System configuration files.
driver.obj	The compiled or assembled code for your device drivers. This can be a library of procedures.
IOOPT1.LIB	I/O System options library.
IOS.LIB	I/O System library.
RPIFC.LIB	Interface library.

Refer to the iRMX 86 CONFIGURATION GUIDE for a complete description of the I/O System configuration process.

APPENDIX A. COMMON DRIVER SUPPORT ROUTINES

This appendix describes, in general terms, the operations of the common device driver support routines. The routines described include:

INIT$IO
FINISH$IO
QUEUE$IO
CANCEL$IO
INTERRUPT$TASK

These routines are supplied with the I/O System and are the device driver routines actually called when an application task makes an I/O request of a common device. These routines ultimately call the user-written device initialize, device finish, device start, device stop, and device interrupt procedures.

This appendix provides descriptions of these routines in order to show you the steps that an actual device driver follows. You can use this appendix to get a better understanding of the I/O System-supplied portion of a device driver in order to make writing the device-dependent portion easier (the random access driver support routines follow essentially the same pattern). Or you can use it as a guideline for writing custom device drivers.

INIT$IO PROCEDURE

The iRMX 86 I/O System calls INIT$IO when an application task makes an RQAPHYSICAL$ATTACH$DEVICE system call and there are no units of the device currently attached. The iRMX 88 I/O System calls INIT$IO when an application task attaches or creates a file on the device and no other files on the device are attached.

INIT$IO initializes objects used by the remainder of the driver routines, creates an interrupt task, and calls a user-supplied procedure to initialize the device itself.

When the I/O System calls INIT$IO, it passes the following parameters:

- A pointer to the DUIB of the device-unit to initialize

- In the iRMX 86 environment, a pointer to the location where INIT$IO must return a token for a data segment (data storage area) that it creates

- A pointer to the location where INIT$IO must return the condition code

The following paragraphs show the general steps that the INIT$IO procedure goes through in order to initialize the device. Figure A-1 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

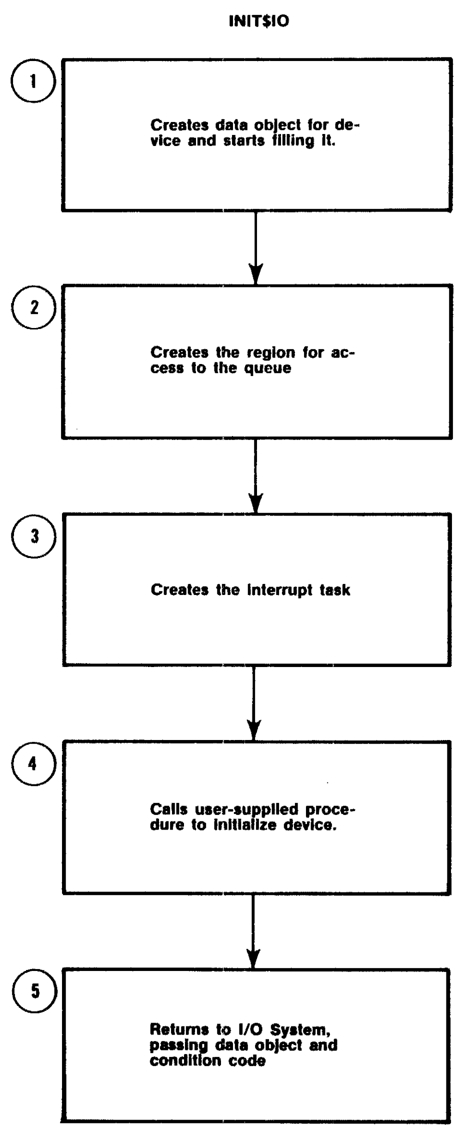

Figure A-1. Common Device Driver Initialize I/O Procedure

1. It creates a data storage area which will be used by all of the procedures in the device driver. The size of this area depends in part on the number of units in the device and any special space requirements of the device. INIT$IO then begins initializing this area and eventually places the following information there:

   - The value of the DS (data segment) register

   - A token (identifier) for a region (exchange --- for mutual exclusion)

   - The address of the DUIB for this device-unit

   - A token (identifier) for the interrupt task

   - Other values indicating that the queue is empty and the segment is not busy

   INIT$IO also reserves space in the data storage area for device data.

2. It creates a region. The other procedures of the device driver gain access from this region whenever they place a request on the queue or remove a request from the queue. INIT$IO places a token for this region in the data object.

3. It creates an interrupt task. This interrupt task handles the interrupts generated by the device for which INIT$IO was called. INIT$IO places a token for this task in the data storage area.

4. It calls a user-written device initialization procedure that initializes the device itself. It gets the address of this procedure by examining the device information table portion of the DUIB. Refer to Chapter 3 for information on how to write this initialization procedure.

5. It returns control to the I/O System, passing a token for the data storage area and a condition code which indicates the success of the initialize operation.

FINISH$IO PROCEDURE

The iRMX 86 I/O System calls FINISH$IO when an application task makes an RQAPHYSICAL$DETACH$DEVICE system call and there are no other units of the device currently attached. The iRMX 88 I/O System calls FINISH$IO when an application detaches or deletes a file and no other files on the device are attached.

FINISH$IO deletes the objects used by the other device driver routines, deletes the interrupt task, and calls a user-supplied procedure to perform final processing on the device itself.

When the I/O System calls FINISH$IO, it passes the following parameters:

- A pointer to the DUIB of the device-unit just detached
- A pointer to the data storage area created by INIT$IO

The following paragraphs show the general steps that the FINISH$IO procedure goes through in order to terminate processing for a device. Figure A-2 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

1. It calls a user-written device finish procedure that performs any necessary final processing on the device itself. FINISH$IO gets the address of this procedure by examining the device information table portion of the DUIB. Refer to the Chapter 4 for information about device information tables.

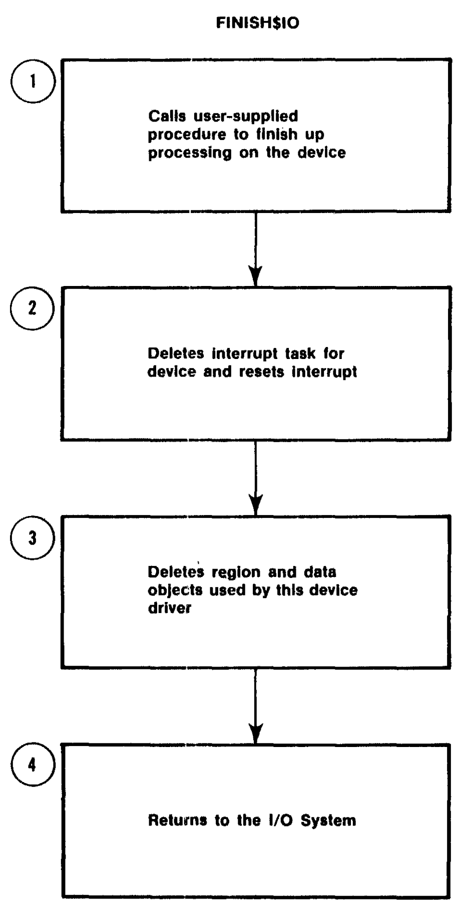

Figure A-2. Common Device Driver Finish I/O Procedure

2. It deletes the interrupt task originally created for the device by the INIT$IO procedure and cancels the assignment of the interrupt handler to the specified interrupt level.

3. It deletes the region and the data storage area originally created by the INIT$IO procedure, allowing the operating system to reallocate the memory used by these objects.

4. It returns control to the I/O System.

QUEUE$IO PROCEDURE

The I/O System calls the QUEUE$IO procedure in order to place an I/O request on a queue of requests. This queue has the structure of the doubly linked list shown in Figure 2-2. If the device itself is not busy, QUEUE$IO also starts the request.

When the I/O System calls QUEUE$IO, it passes the following parameters

- A token (identifier) for the IORS
- A pointer to the DUIB
- A token (identifier) for the data object originally created by INIT$IO

The following paragraphs show the general steps that the QUEUE$IO procedure goes through in order to place a request on the I/O queue. Figure A-3 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

1. It sets the DONE field in the IORS to 0H, indicating that the request has not yet been completely processed. Other procedures that start the I/O transfers and handle interrupt processing also examine and set this field.

2. It receives access to the queue from the region. This allows QUEUE$IO to adjust the queue without concern that other tasks might also be doing this at the same time.

3. It places the IORS on the queue.

4. It calls an I/O System-supplied procedure in order to start the processing of the request. This results in a call to a user-written device start procedure which actually sends the data to the device itself. This start procedure is described in Chapter 5. If the device is already busy processing some other request, this step does not start the data transfer.

5. It surrenders access to the queue, allowing other routines to insert or remove requests from the queue.

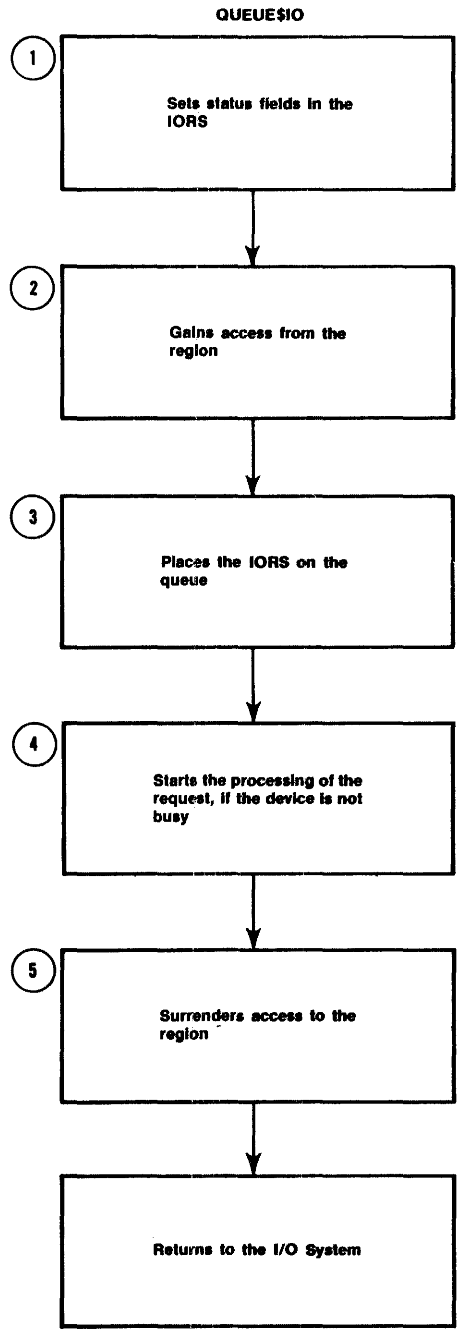

Figure A-3. Common Device Driver Queue I/O Procedure

CANCEL$IO PROCEDURE

The I/O System calls CANCEL$IO to remove one or more requests from the queue and possibly to stop the processing of a request, if it has already been started. The iRMX 86 I/O System calls this procedure in one of two instances:

- If a user makes an RQAPHYSICAL$DETACH$DEVICE system call and specifies the hard detach option (refer to the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL for information about this system call). The hard detach removes all requests from the queue.

- If the job containing the task that makes an I/O request is deleted. In this case, the I/O System calls CANCEL$IO to remove all of that task's requests from the queue.

When the I/O System calls CANCEL$IO, it passes the following parameters:

- An id value that identifies requests to be cancelled

- A pointer to the DUIB

- A token (identifier) for the device data storage area

The following paragraphs show the general steps that the CANCEL$IO procedure goes through in order to cancel an I/O request. Figure A-4 illustrates these steps. The numbers in the figure correspond to the step numbers in the text.

1. It receives access to the queue from the region. This allows it to remove requests from the queue without concern that other tasks might also be processing the IORS at the same time.

2. It locates a request that is to be cancelled by looking at the cancel$id field of the queued IORSs, starting at the front of the queue.

3. If the request that is to be cancelled is at the head of the queue, that is, the device is processing the request, CANCEL$IO calls a user-written device stop procedure that stops the device from further processing. Refer to the Chapter 5 for information on how to write this device stop procedure.

4. If the request is finished, or if the IORS is not at the head of the queue, CANCEL$IO removes the IORS from the queue and sends it to the response mailbox (exchange) indicated in the IORS.

5. It surrenders access to the queue, allowing other procedures to insert or remove requests from the queue.

NOTE

> The additional CLOSE request supplied by the I/O System will not be processed until all other requests with the given cancel$id value have been dealt with.

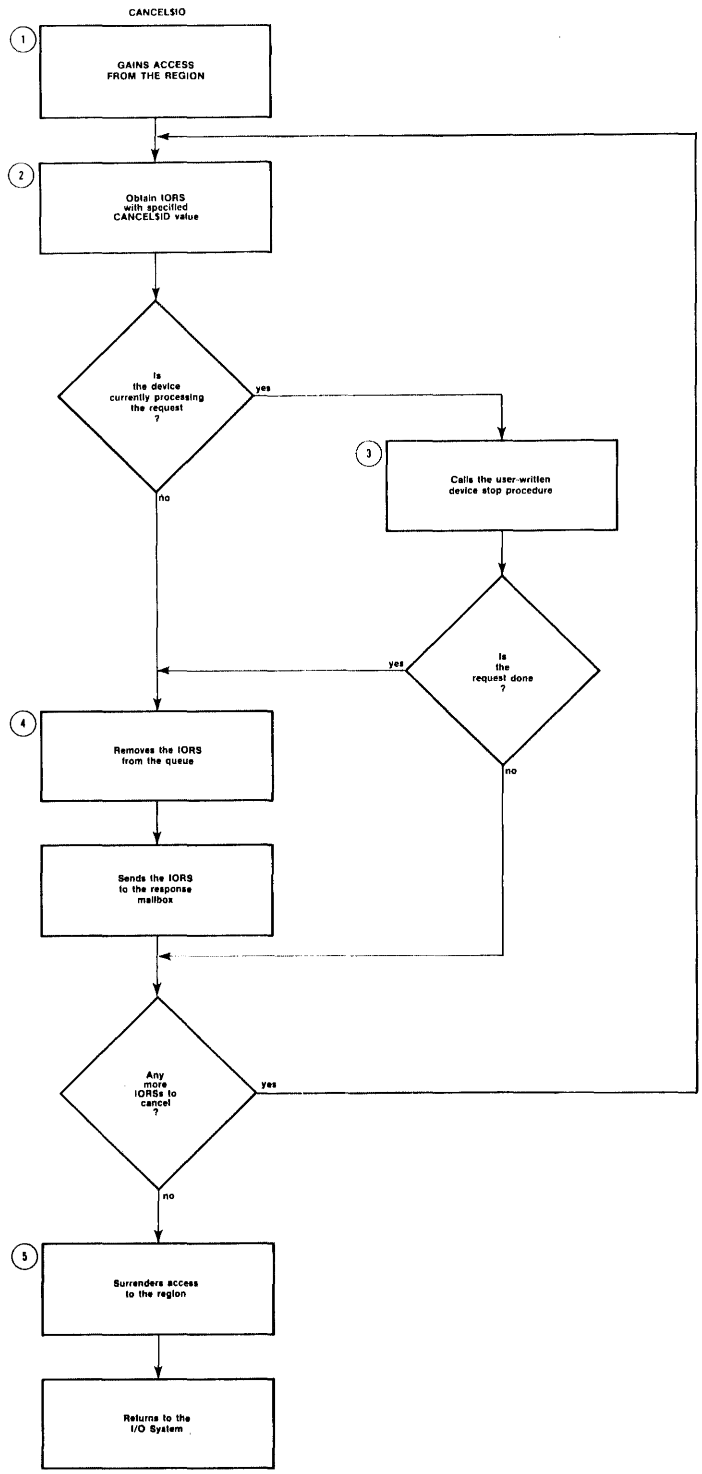

Figure A-4. Common Device Driver Cancel I/O Procedure

INTERRUPT TASK (INTERRUPT$TASK)

As a part of its processing, the INIT$IO procedure creates an interrupt task for the entire device. This interrupt task responds to all interrupts generated by the units of the device, processes those interrupts, and starts the device working on the next I/O request on the queue.

The following paragraphs show the general steps that the interrupt task for the common device driver goes through in order to process a device interrupt. Figure A-5 illustrates these steps. The numbers in Figure A-5 correspond to the step numbers in the text.

1. It uses the contents of the iAPX 86 DS register to obtain a token (identifier) for the device data storage area. This is possible because of the following two reasons:

   - When INIT$IO created the interrupt task, instead of specifying the correct contents of the DS register, it passed the address of the data object as the contents of the task's DS register.

   - When the INIT$IO procedure created the data storage area, it included the correct contents of the DS register in one of the fields.

   When the interrupt task starts running, it saves the contents of the DS register (to use as the address of the data storage area) and sets the DS register to the value listed in the field of the data storage area. Thus the task has the correct value in its DS register and it has the address of the data storage area. This is the mechanism that is used to pass the address of the device's data storage area from the INIT$IO procedure to the interrupt task.

2. It makes an RQSETINTERRUPT system call to indicate that it is an interrupt task associated with the interrupt handler supplied with the common device driver. It also indicates the interrupt level to which it will respond.

3. It begins an infinite loop by waiting for an interrupt of the specified level.

4. Via a region, it gains access to the request queue. This allows it to examine the first entry in the request queue without concern that other tasks are modifying it at the same time.

5. It calls a user-written device-interrupt procedure to process the actual interrupt. This can involve verifying that the interrupt was legitimate or any other operation that the device requires. This interrupt procedure is described further in Chapter 3.

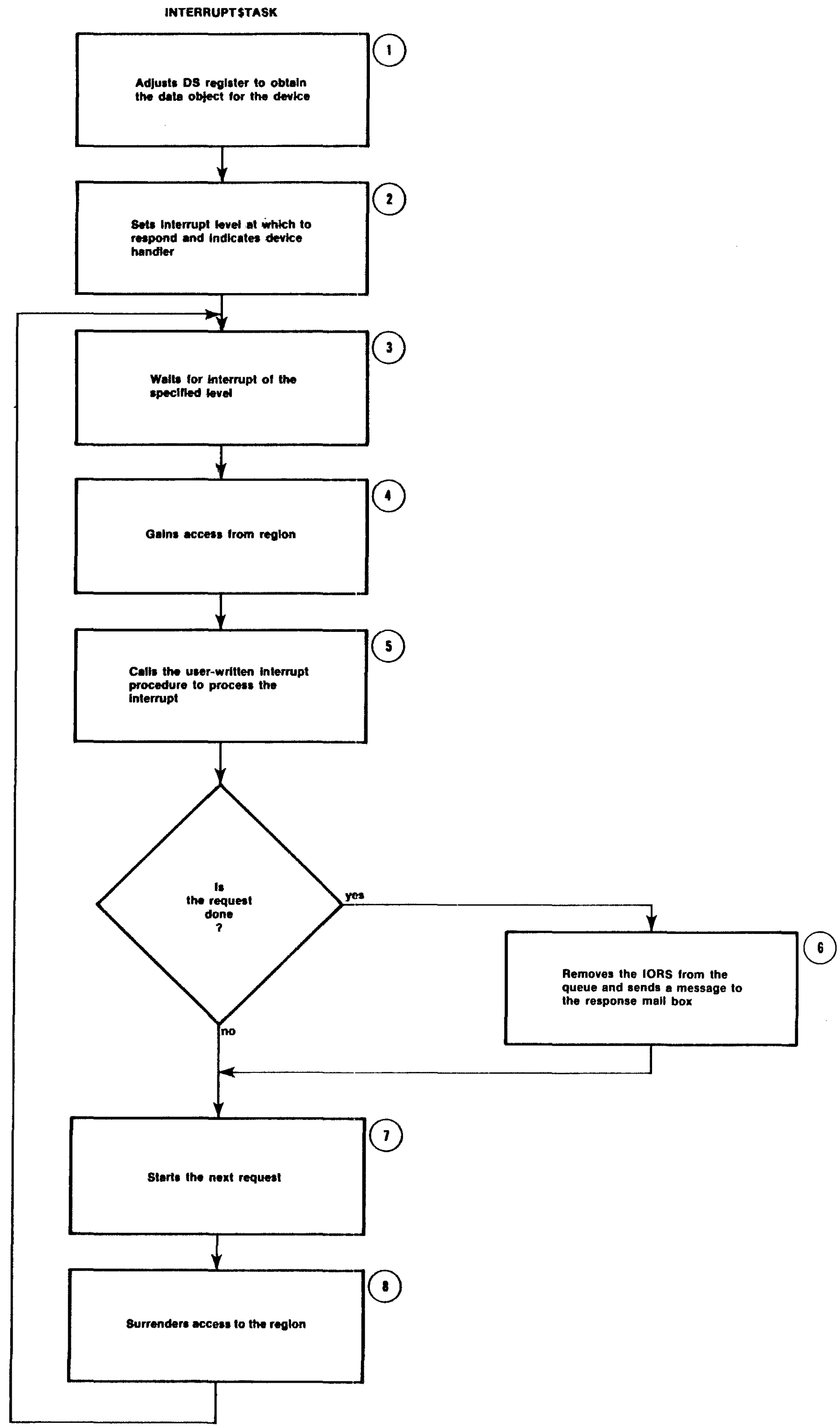

Figure A-5. Common Device Driver Interrupt Task

6. If the request has been completely processed, (one request may require multiple reads or writes, for example), the interrupt task removes the IORS from the queue and sends it as a message to the response mailbox (exchange) indicated in the IORS. If the request is not completely processed, the interrupt task leaves the IORS at the head of the queue.

7. If there are requests on the queue, the interrupt task initiates the processing of the next I/O request.

8. In any case, the interrupt task then surrenders access to the queue, allowing other routines to modify the queue, and loops back to wait for another interrupt.

APPENDIX B. EXAMPLES OF DEVICE DRIVERS

This appendix contains three examples of device drivers. The first, a common driver, is a driver for a box with eight lights and eight switches. The second, also a common driver, drives a line printer. And the third, a random access driver, is a driver for the iSBC 206 disk controller.

Note that the names of the procedures in the examples are not device$start, device$interrupt, etc., as in the text of this manual. This is because the actual names are placed, during configuration, in the appropriate DUIBs.

```
PL/M-86 COMPILER    LIGHT

SERIES-III PL/M-86 DEBUG X119 COMPILATION OF MODULE LIGHT
OBJECT MODULE PLACED IN :F5:LIGHT.OBJ
COMPILER INVOKED BY:  PLM86.86 :F5:LIGHT.P86 ROM COMPACT


   1              LIGHT:
                  /*********************************************************************

                          This driver is written to control a light/switch box
                      attached to an iSBC508 I/O Expansion Board. The box consists of
                      a series of 8 LED's (one for each bit) and 8 switches which
                      allow a byte to be 'read' from the device.  The box is attached
                      to the board at port 0, and an interrupt level of the user's
                      choosing (set at configuration time in the DUIB of the IOS) is
                      triggered by a debouncing circuit attached to the appropriate
                      interrupt level on the Multibus.

                          When the attachment is made to this device by a call to
                      RQ$A$PHYSICAL$ATTACH$DEVICE, the box will light all LED's to
                      indicate a successful attachment.  When the device is detached,
                      all LED's will be turned off.  Anytime a read or write is done
                      to or from the device, the interrupt must be manually triggered
                      by the user to indicate that the device has successfully
                      completed the transfer.

                          In order to accomplish this, the device was treated as a
                      common device, thereby allowing the use of the default routines
                      init$io, queue$io, finish$io, and cancel$io.  In addition, the
                      Intel supplied procedures default$stop and default$finish were
                      used, since no action was required of the device on any of
                      these procedures.

                          This device and driver combination are not intended to be
                      used in a practical application, but rather are meantto show
                      the versatility and configurability of the device driver and to
                      present a simple example of one.

                  *********************************************************************/

                    DO;

   2    1             DECLARE TRUE              LITERALLY   '0FFH';
   3    1             DECLARE FALSE             LITERALLY   '  0H';
   4    1             DECLARE E$OK              LITERALLY   '  0H';
   5    1             DECLARE E$IDDR            LITERALLY   ' 2AH';
   6    1             DECLARE E$IO              LITERALLY   ' 2BH';
   7    1             DECLARE F$READ            LITERALLY   '   0';
   8    1             DECLARE F$WRITE           LITERALLY   '   1';
   9    1             DECLARE F$SEEK            LITERALLY   '   2';
  10    1             DECLARE F$SPECIAL         LITERALLY   '   3';
  11    1             DECLARE F$ATTACH$DEV      LITERALLY   '   4';
  12    1             DECLARE F$DETACH$DEV      LITERALLY   '   5';
  13    1             DECLARE F$OPEN            LITERALLY   '   6';
  14    1             DECLARE F$CLOSE           LITERALLY   '   7';

  15    1             DECLARE ALL$LIGHTS$OFF LITERALLY      '00000000B';
```

```
PL/M-86 COMPILER    LIGHT

  16   1          DECLARE ALL$LIGHTS$ON  LITERALLY   '11111111B';

  17   1      SET$LIGHTS:
              /* Routine to output the corresponding string to the light box */
                PROCEDURE(PORT,NEW$VALUE)REENTRANT;

  18   2          DECLARE PORT           WORD;
  19   2          DECLARE NEW$VALUE      BYTE;

  20   2          OUTPUT(PORT)=NEW$VALUE;

  21   2        END SET$LIGHTS;

  22   1      READ$SWITCHES:
              /* Routine to read the switches on the front panel of the
                      light box */
                PROCEDURE(PORT) BYTE REENTRANT;

  23   2          DECLARE PORT           WORD;

  24   2          RETURN( INPUT(PORT) );

  25   2        END READ$SWITCHES;
```

```
                $EJECT

 26   1         LIGHT$BOX$INIT$IO:
                  PROCEDURE (DUIB$PTR,DDATA$PTR,STATUS$PTR ) REENTRANT PUBLIC;

                /*********************************************************************

                       This procedure will establish a connection to the lights by
                    turning off all lights on all attached devices (as determined in
                    the num$units in the common$device$info block).

                *********************************************************************/
 27   2              DECLARE DUIB$PTR        POINTER;
 28   2              DECLARE DUIB BASED DUIB$PTR STRUCTURE (
                       NAME(14)              BYTE,
                       FILE$DRIVERS          WORD,
                       FUNCTS                BYTE,
                       FLAGS                 BYTE,
                       DEV$GRAN              WORD,
                       LOW$DEV$SIZE          WORD,
                       HIGH$DEV$SIZE         WORD,
                       DEVICE                BYTE,
                       UNIT                  BYTE,
                       DEV$UNIT              WORD,
                       INIT$IO               WORD,
                       FINISH$IO             WORD,
                       QUEUE$IO              WORD,
                       CANCEL$IO             WORD,
                       DEVICE$INFO$PTR       POINTER,
                       UNIT$INFO$PTR         POINTER,
                       UPDATE$TIMEOUT        WORD,
                       NUM$BUFFERS           WORD,
                       PRIORITY              BYTE);
 29   2             DECLARE STATUS$PTR       POINTER;
 30   2             DECLARE DDATA$PTR        POINTER;
 31   2             DECLARE COMMON$DEVICE$INFO$PTR POINTER;
 32   2             DECLARE COMMON$DEVICE$INFO BASED
                            COMMON$DEVICE$INFO$PTR STRUCTURE(
                       LEVEL                 WORD,
                       PRIORITY              BYTE,
                       STACK$SIZE            WORD,
                       DATA$SIZE             WORD,
                       NUM$UNITS             WORD,
                       DEVICE$INIT           WORD,
                       DEVICE$FINISH         WORD,
                       DEVICE$START          WORD,
                       DEVICE$STOP           WORD,
                       DEVICE$INTERRUPT      WORD,
                       BASE                  WORD);
 33   2             DECLARE INDEX            WORD;

 34   2             COMMON$DEVICE$INFO$PTR=DUIB.DEVICE$INFO$PTR;
 35   2             DO INDEX=0 TO (COMMON$DEVICE$INFO.NUM$UNITS-1);

 36   3                 CALL SET$LIGHTS ((COMMON$DEVICE$INFO.BASE + INDEX),
                                         ALL$LIGHTS$OFF);
```

```
PL/M-86 COMPILER    LIGHT

 37   3           END;
 38   2       END LIGHT$BOX$INIT$IO;


 39   1       LIGHT$BOX$START$IO:
                PROCEDURE (IORS$PTR,DUIB$PTR,DDATA$PTR) REENTRANT PUBLIC;

 40   2            DECLARE DUIB$PTR         POINTER;
 41   2            DECLARE DUIB BASED DUIB$PTR STRUCTURE (
                     NAME(14)                BYTE,
                     FILE$DRIVERS            WORD,
                     FUNCTS                  BYTE,
                     FLAGS                   BYTE,
                     DEV$GRAN                WORD,
                     LOW$DEV$SIZE            WORD,
                     HIGH$DEV$SIZE           WORD,
                     DEVICE                  BYTE,
                     UNIT                    BYTE,
                     DEV$UNIT                WORD,
                     INIT$IO                 WORD,
                     FINISH$IO               WORD,
                     QUEUE$IO                WORD,
                     CANCEL$IO               WORD,
                     DEVICE$INFO$PTR         POINTER,
                     UNIT$INFO$PTR           POINTER,
                     UPDATE$TIMEOUT          WORD,
                     NUM$BUFFERS             WORD,
                     PRIORITY                BYTE);
 42   2           DECLARE DDATA$PTR        POINTER;
 43   2           DECLARE COMMON$DEVICE$INFO$PTR POINTER;
 44   2           DECLARE COMMON$DEVICE$INFO BASED
                             COMMON$DEVICE$INFO$PTR STRUCTURE(
                     LEVEL                   WORD,
                     PRIORITY                BYTE,
                     STACK$SIZE              WORD,
                     DATA$SIZE               WORD,
                     NUM$UNITS               WORD,
                     DEVICE$INIT             WORD,
                     DEVICE$FINISH           WORD,
                     DEVICE$START            WORD,
                     DEVICE$STOP             WORD,
                     DEVICE$INTERRUPT        WORD,
                     BASE                    WORD);
 45   2           DECLARE IORS$PTR         POINTER;
 46   2           DECLARE IORS BASED IORS$PTR STRUCTURE(
                     STATUS                  WORD,
                     UNIT$STATUS             WORD,
                     ACTUAL                  WORD,
                     ACTUAL$FILL             WORD,
                     DEVICE                  WORD,
                     UNIT                    BYTE,
                     FUNCT                   BYTE,
                     SUBFUNCT                WORD,
                     LOW$DEV$LOC             WORD,
                     HIGH$DEV$LOC            WORD,
                     BUFF$PTR                POINTER,
```

```
PL/M-86 COMPILER    LIGHT

                        COUNT                   WORD,
                        COUNT$FILL              WORD,
                        AUX$PTR                 POINTER,
                        LINK$FOR                POINTER,
                        LINK$BACK               POINTER,
                        RESP$MBOX               WORD,
                        DONE                    BYTE,
                        FILL                    BYTE,
                        CANCEL$ID               WORD );

                 /* Initialize the I/O Structures  */

 47   2            COMMON$DEVICE$INFO$PTR=DUIB.DEVICE$INFO$PTR;
 48   2            IORS.STATUS=E$IDDR;
 49   2            IORS.ACTUAL=0;
 50   2            IORS.DONE=TRUE;

                 /* Check for valid I/O functions  */

 51   2            IF (IORS.FUNCT <= F$CLOSE) THEN

                 /* I/O function is valid, go ahead */

 52   2               DO CASE (IORS.FUNCT);

                     /* Read-- Set done to false, since function will be finished
                               by interrupt routine.  Set status to E$OK, since
                               function is valid.   */

 53   3                  DO;
 54   4                    IORS.DONE=FALSE;
 55   4                    IORS.STATUS=E$OK;
 56   4                  END;

                     /* Write-- Set done to false, since function will be finished
                               by interrupt routine.  Set status to E$OK, since
                               function is valid.   */

 57   3                  DO;
 58   4                    IORS.DONE=FALSE;
 59   4                    IORS.STATUS=E$OK;
 60   4                  END;

                    /* Seek-- Function is invalid, return E$IDDR */

 61   3                  DO;
 62   4                  END;

                    /* Special-- Function is invalid, return E$IDDR */

 63   3                  DO;
 64   4                  END;

                    /* Attach-- Activate all lights, return E$OK */

 65   3                  DO;
 66   4                    CALL SET$LIGHTS ((COMMON$DEVICE$INFO.BASE + DUIB.UNIT),
```

PL/M-86 COMPILER   LIGHT

```
                                        ALL$LIGHTS$ON);
 67    4                      IORS.STATUS=E$OK;
 68    4                    END;

                        /* Detach-- Deactivate all lights, return E$OK. */

 69    3                    DO;
 70    4                      CALL SET$LIGHTS ((COMMON$DEVICE$INFO.BASE + DUIB.UNIT),
                                        ALL$LIGHTS$OFF);
 71    4                      IORS.STATUS=E$OK;
 72    4                    END;

                       /* Open-- Valid function, return E$OK */

 73    3                    DO;
 74    4                      IORS.STATUS=E$OK;
 75    4                    END;

                       /* Close-- Valid function, return E$OK */

 76    3                    DO;
 77    4                      IORS.STATUS=E$OK;
 78    4                    END;

 79    3                 END; /* case */
 80    2             END LIGHT$BOX$START$IO;
```

```
PL/M-86 COMPILER    LIGHT

              $EJECT

 81   1       LIGHT$BOX$INTERRUPT:
                PROCEDURE (IORS$PTR,DUIB$PTR,DDATA$PTR) REENTRANT PUBLIC;

 82   2           DECLARE DUIB$PTR          POINTER;
 83   2           DECLARE DUIB BASED DUIB$PTR STRUCTURE (
                    NAME(14)                BYTE,
                    FILE$DRIVERS            WORD,
                    FUNCTS                  BYTE,
                    FLAGS                   BYTE,
                    DEV$GRAN                WORD,
                    LOW$DEV$SIZE            WORD,
                    HIGH$DEV$SIZE           WORD,
                    DEVICE                  BYTE,
                    UNIT                    BYTE,
                    DEV$UNIT                WORD,
                    INIT$IO                 WORD,
                    FINISH$IO               WORD,
                    QUEUE$IO                WORD,
                    CANCEL$IO               WORD,
                    DEVICE$INFO$PTR         POINTER,
                    UNIT$INFO$PTR           POINTER,
                    UPDATE$TIMEOUT          WORD,
                    NUM$BUFFERS             WORD,
                    PRIORITY                BYTE);
 84   2           DECLARE DDATA$PTR         POINTER;
 85   2           DECLARE COMMON$DEVICE$INFO$PTR POINTER;
 86   2           DECLARE COMMON$DEVICE$INFO BASED
                            COMMON$DEVICE$INFO$PTR STRUCTURE(
                    LEVEL                   WORD,
                    PRIORITY                BYTE,
                    STACK$SIZE              WORD,
                    DATA$SIZE               WORD,
                    NUM$UNITS               WORD,
                    DEVICE$INIT             WORD,
                    DEVICE$FINISH           WORD,
                    DEVICE$START            WORD,
                    DEVICE$STOP             WORD,
                    DEVICE$INTERRUPT        WORD,
                    BASE                    WORD);
 87   2           DECLARE IORS$PTR          POINTER;
 88   2           DECLARE IORS BASED IORS$PTR STRUCTURE (
                    STATUS                  WORD,
                    UNIT$STATUS             WORD,
                    ACTUAL                  WORD,
                    ACTUAL$FILL             WORD,
                    DEVICE                  WORD,
                    UNIT                    BYTE,
                    FUNCT                   BYTE,
                    SUBFUNCT                WORD,
                    LOW$DEV$LOC             WORD,
                    HIGH$DEV$LOC            WORD,
                    BUFF$PTR                POINTER,
                    COUNT                   WORD,
                    COUNT$FILL              WORD,
```

```
PL/M-86 COMPILER    LIGHT

                        AUX$PTR                     POINTER,
                        LINK$FOR                    POINTER,
                        LINK$BACK                   POINTER,
                        RESP$MBOX                   WORD,
                        DONE                        BYTE,
                        FILL                        BYTE,
                        CANCEL$ID                   WORD );
  89    2           DECLARE BUFFER$PTR          POINTER;
  90    2           DECLARE BUFFER BASED BUFFER$PTR (1) BYTE;

                   /*  Check for a valid interrupt */

  91    2           IF (IORS$PTR<>0) THEN
  92    2             DO;

  93    3               COMMON$DEVICE$INFO$PTR=DUIB.DEVICE$INFO$PTR;
  94    3               BUFFER$PTR=IORS.BUFF$PTR;

  95    3                 DO CASE (IORS.FUNCT);

                             /* Read-- Bring in switch reading */

  96    4                   BUFFER (IORS.ACTUAL)=READ$SWITCHES(
                                            COMMON$DEVICE$INFO.BASE + DUIB.UNIT);

                             /* Write-- Output light pattern */

  97    4                   CALL SET$LIGHTS ((COMMON$DEVICE$INFO.BASE + DUIB.UNIT),
                                                                BUFFER (IORS.ACTUAL));

  98    4                 END;
  99    3               IORS.ACTUAL=IORS.ACTUAL+1;
 100    3               IF (IORS.ACTUAL=IORS.COUNT) THEN
 101    3                 DO;
 102    4                    IORS.STATUS=E$OK;
 103    4                    IORS.DONE=TRUE;
 104    4                 END;
 105    3           END;
 106    2         END LIGHT$BOX$INTERRUPT;
 107    1       END LIGHT;


MODULE INFORMATION:

     CODE AREA SIZE     = 021BH     539D
     CONSTANT AREA SIZE = 0000H       0D
     VARIABLE AREA SIZE = 0000H       0D
     MAXIMUM STACK SIZE = 001EH      30D
     377 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER     iprntr.p86
                     printer$start$interrupt


SERIES-III PL/M-86 DEBUG X119 COMPILATION OF MODULE IPRNTR
OBJECT MODULE PLACED IN :F1:IPRNTR.OBJ
COMPILER INVOKED BY:  PLM86.86 :F1:IPRNTR.P86 COMPACT ROM NOTYPE OPTIMIZE(3)


                 $title ('iprntr.p86')
                 /*
                  *  iprntr.p86
                  *
                  *     This module implements centronix-type interface line printer
                  *     driver.  It is written as a 'common' device driver.  It is
                  *     assumed that the reader is familiar with the 8255 chip.
                  *
                  *  LANGUAGE DEPENDENCIES:
                  *     COMPACT ROM OPTIMIZE(3)
                  */
                 $include(:fl:icpyrt.not)
              =  /*
              =   *  INTEL CORPORATION PROPRIETARY INFORMATION.  THIS LISTING IS
              =   *  SUPPLIED UNDER THE TERMS OF A LICENSE AGREEMENT WITH INTEL
              =   *  CORPORATION AND MAY NOT BE COPIED NOR DISCLOSED EXCEPT IN
              =   *  ACCORDANCE WITH THE TERMS OF THAT AGREEMENT.
              =   */
   1             iprntr: DO;
                 $include(:fl:icomon.lit)
              =  $save nolist
                 $include(:fl:iparam.lit)
              =  $save nolist
                 $include(:fl:inutyp.lit)
              =  $save nolist
                 $include(:fl:iiors.lit)
              =  $save nolist
                 $include(:fl:iduib.lit)
              =  $save nolist
                 $include(:fl:iprntr.lit)
              =      /*
              =       * Common device driver information
              =       *
              =       * level:                Interrupt level
              =       * priority:             Priority of interrupt task
              =       * stack$size:           Stack size for interrupt task
              =       * data$size:            Device local data size
              =       * num$units:            Number of units on device
              =       * device$init:          Init device procedure
              =       * device$finish:        Finished with device procedure
              =       * device$start:         Start device procedure
              =       * device$stop:          Stop device procedure
              =       * device$interrupt:     Device interrupt procedure
              =       */

  13     1    =  DECLARE COMMON$DEV$INFO LITERALLY '
              =      level                  WORD,
              =      priority               BYTE,
              =      stack$size             WORD,
              =      data$size              WORD,
```

```
PL/M-86 COMPILER     iprntr.p86
                     printer$start$interrupt

           =      num$units            WORD,
           =      device$init          WORD,
           =      device$finish        WORD,
           =      device$start         WORD,
           =      device$stop          WORD,
           =      device$interrupt     WORD';

  14   1   =  DECLARE i8255$INFO LITERALLY '
           =      A$port               WORD,
           =      B$port               WORD,
           =      C$port               WORD,
           =      Control$port         WORD';

  15   1   =  DECLARE
           =      PRINTER$DEVICE$INFO LITERALLY 'STRUCTURE(
           =          COMMON$DEV$INFO,
           =          i8255$INFO,
           =          tab$control   WORD)';
              $include(:f1:i8255.lit)
           =  /*
           =   *  8255 is programmed as follows:
           =   *
           =   *     Group A:  Mode 0
           =   *     Group B:  Mode 1
           =   *
           =   *     Port A and Upper Port C: OUTPUT
           =   *     Port B and Lower Port C: INPUT
           =   *
           =   *  Port C definition (bit 0 is LSB;  bit 7 is MSB):
           =   *
           =   *       Bit  0    -   Interrupt to CPU (not used by the driver)
           =   *            1    -   Character acknowledge from the printer
           =   *            2    -   Printer interrupt enable
           =   *            3    -   Paper error status (not used by the driver)
           =   *            4    -   Character strobe to the printer
           =   *        5,6,7    -   not used
           =   */
  16   1   =  DECLARE
           =     MODE$WORD      LITERALLY    '87H',
           =     CHAR$ACK       LITERALLY    '02H',
           =     INT$ENABLE     LITERALLY    '05H',
           =     INT$DISABLE    LITERALLY    '04H',
           =     STROBE$ON      LITERALLY    '09H',
           =     STROBE$OFF     LITERALLY    '08H';
              $include(:f1:iprerr.lit)
           =  $save nolist
              /*
               *  literal declaration
               */
  18   1      DECLARE
                 TAB$CHAR LITERALLY '09H',
                 SPACE    LITERALLY '20H';
```

```
PL/M-86 COMPILER    iprntr.p86
                    printer$start$interrupt

                $eject
                $subtitle('printer$start$interrupt')
                /*
                 *  printer$start/printer$interrupt
                 *       start/interrupt procedure for the line printer
                 *
                 *  CALLING SEQUENCE:
                 *       CALL printer$start$interrupt (iors$p, duib$p, ddata$p);
                 *
                 *  INTERFACE VARIABLES:
                 *       iors$p   -   I/O request/result segment pointer
                 *       duib$p   -   pointer to the device-unit info. block
                 *       ddata$p  -   pointer to the device(printer) data segment.
                 *
                 *  CALLS:  None
                 *
                 */

 19   1         printer$start$interrupt: PROCEDURE (iors$p, duib$p, ddata$p)
                                                         PUBLIC REENTRANT;
 20   2            DECLARE
                      (iors$p, duib$p, ddata$p)   POINTER;
 21   2            DECLARE
                      iors     BASED  iors$p   IO$REQ$RES$SEG,
                      duib     BASED  duib$p   DEV$UNIT$INFO$BLOCK;
 22   2            DECLARE
                      dinfo$p  POINTER,
                      dinfo    BASED  dinfo$p PRINTER$DEVICE$INFO;
 23   2            DECLARE
                       buffer$p     POINTER,
                      (char  BASED  buffer$p)(1) BYTE;

 24   2            dinfo$p = duib.device$info$p;

                   /*
                    *  test for spurious interrupts
                    */
 25   2            IF iors$p = 0 THEN
 26   2            DO;
                      /*
                       * turn off the interrupt and return
                       */
 27   3               OUTPUT(dinfo.Control$port) = INT$DISABLE;
 28   3               RETURN;
 29   3            END;

 30   2            DO CASE (iors.funct);

                      /* read */
 31   3               DO;
 32   4                  iors.status = E$IDDR;
 33   4                  iors.done = TRUE;
 34   4               END;

                      /* write */
 35   3               DO;
```

```
PL/M-86 COMPILER      iprntr.p86
                      printer$start$interrupt

                          /* get the buffer pointer */
  36    4                 buffer$p = iors.buff$p;

                          /* disable printer interrupt */
  37    4                 OUTPUT(dinfo.Control$port) = INT$DISABLE;

  38    4                 DO WHILE (iors.actual < iors.count);

                              /*
                               *  convert TAB character to a SPACE character if the
                               *  printer does not handle them
                               */
  39    5                     IF ((char(iors.actual) = TAB$CHAR) AND
                                                      ((dinfo.tab$control) = FALSE))
                                  THEN char(iors.actual) = SPACE;
                              /*
                               * 1's complement the character and send it to the
                               * printer.  Port-A is the data port
                               */
  41    5                     OUTPUT(dinfo.A$port) = NOT(char(iors.actual));
                              /*
                               * strobe the line printer
                               * this is a way of telling the printer that there is
                               * valid data on the bus
                               */
  42    5                     OUTPUT(dinfo.Control$port) = STROBE$ON;
  43    5                     OUTPUT(dinfo.Control$port) = STROBE$OFF;
                              /*
                               *  increment the count of chars printed
                               */
  44    5                     iors.actual = iors.actual + 1;
                              /*
                               *  test whether printer acknowledgement bit is set
                               */
  45    5                     IF (INPUT(dinfo.C$port) AND CHAR$ACK) = 0 THEN
  46    5                     DO;
                                  /*
                                   *  printer didn't acknowledge. Hopefully it has
                                   *  started printing. So enable the printer interrupt
                                   *  and return(printer will interrupt when it's done)
                                   */
  47    6                         OUTPUT(dinfo.Control$port) = INT$ENABLE;
  48    6                         RETURN;
  49    6                     END;
                              ELSE
  50    5                     DO;
                                  /*
                                   *  printer copied the character into its buffer
                                   *  clear printer acknowledge bit by reading port B.
                                   *  actual$fill field in the iors is used as a tempo-
                                   *  rary variable. Char read is ignored.
                                   */
  51    6                         iors.actual$fill = INPUT(dinfo.B$port);
  52    6                     END;

  53    5                 END;  /* end of DO WHILE statement */
```

```
PL/M-86 COMPILER     iprntr.p86
                     printer$start$interrupt

                         /*
                          *  set iors.done to TRUE
                          *  set iors.status to OK
                          */
 54   4                  iors.status = E$OK;
 55   4                  iors.done = TRUE;
 56   4              END;

                     /* seek */
 57   3              DO;
 58   4                  iors.status = E$IDDR;
 59   4                  iors.done = TRUE;
 60   4              END;

                     /* special */
 61   3              DO;
 62   4                  iors.status = E$IDDR;
 63   4                  iors.done = TRUE;
 64   4              END;

                     /* attach device */
 65   3              DO;
                         /* initialize the 8255 */
 66   4                  OUTPUT(dinfo.Control$port) = MODE$WORD;
 67   4                  iors.status = E$OK;
 68   4                  iors.done = TRUE;
 69   4              END;

                     /* detach device */
 70   3              DO;
 71   4                  iors.status = E$OK;
 72   4                  iors.done = TRUE;
 73   4              END;

                     /* open */
 74   3              DO;
 75   4                  iors.status = E$OK;
 76   4                  iors.done = TRUE;
 77   4              END;

                     /* close */
 78   3              DO;
 79   4                  iors.status = E$OK;
 80   4                  iors.done = TRUE;
 81   4              END;

 82   3          END;  /* end of DO CASE statement */

 83   2      END printer$start$interrupt;
```

```
PL/M-86 COMPILER    iprntr.p86
                    printer$stop


                $subtitle('printer$stop')
                /*
                 *  printer$stop
                 *       stop procedure for the line printer
                 *
                 *  CALLING SEQUENCE:
                 *       CALL printer$stop (iors$p, duib$p, ddata$p);
                 *
                 *  INTERFACE VARIABLES:
                 *       iors$p    -   I/O request/result segment pointer
                 *       duib$p    -   pointer to the device-unit info. block
                 *       ddata$p   -   pointer to the device(printer) data segment.
                 *
                 *  CALLS:  None
                 *
                 */

 84   1         printer$stop: PROCEDURE (iors$p, duib$p, ddata$p) PUBLIC REENTRANT;
 85   2            DECLARE
                      (iors$p, duib$p, ddata$p)   POINTER;
 86   2            DECLARE
                      iors      BASED  iors$p   IO$REQ$RES$SEG,
                      duib      BASED  duib$p   DEV$UNIT$INFO$BLOCK;
 87   2            DECLARE
                      dinfo$p   POINTER,
                      dinfo     BASED  dinfo$p PRINTER$DEVICE$INFO;

                   /*
                    *  turn off the printer interrupt
                    *  set iors.done to TRUE
                    *  set iors.status to E$OK
                    */

 88   2             dinfo$p = duib.device$info$p;

 89   2             OUTPUT(dinfo.Control$port) = INT$DISABLE;
 90   2             iors.status = E$OK;
 91   2             iors.done = TRUE;

 92   2         END printer$stop;

 93   1         END iprntr;


MODULE INFORMATION:

     CODE AREA SIZE     = 0140H    320D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0016H     22D
     500 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER    i206ds.p86
                    Module Header


SERIES-III PL/M-86 DEBUG X119 COMPILATION OF MODULE I206DS
OBJECT MODULE PLACED IN :F5:I206DS.OBJ
COMPILER INVOKED BY:  PLM86.86 :F5:I206DS.P86 COMPACT NOTYPE OPTIMIZE(3) ROM


              $title('i206ds.p86')
              $subtitle('Module Header')

              /*
               *  i206ds.p86
               *
               *  CONTAINS:
               *      i206$start          maps to device$init.
               *      i206$interrupt      maps to device$interrupt.
               *      i206$init           maps to device$start.
               *
               *  This module contains the procedures that are referenced
               *  in the device information tables.
               *
               * LANGUAGE DEPENDENCIES: COMPACT ROM OPTIMIZE(3)
               */

  1           i206ds: DO;

              $include(:fl:icomon.lit)
          =   $save nolist
              $include(:fl:inutyp.lit)
          =   $save nolist
              $include(:fl:iparam.lit)
          =   $save nolist
              $include(:fl:iiotyp.lit)
          =   $save nolist
              $include(:fl:iiors.lit)
          =   $save nolist
              $include(:fl:iduib.lit)
          =   $save nolist
              $include(:fl:idrinf.lit)
          =   $save nolist
              $include(:fl:i206in.lit)
          =   $save nolist
              $include(:fl:i206dv.lit)
          =   $save nolist
              $include(:fl:iexcep.lit)
          =   $save nolist
              $include(:fl:iioexc.lit)
          =   $save nolist
              $include(:fl:iradsf.lit)
          =   $save nolist

              $include(:fl:i206dp.ext)
          =   $save nolist
              $include(:fl:i206dc.ext)
          =   $save nolist
              $include(:fl:i206fm.ext)
          =   $save nolist




PL/M-86 COMPILER    i206ds.p86
                    Module Header

              $include(:fl:iasmut.ext)
          =   $save nolist
              $include(:fl:inotif.ext)
          =   $save nolist
```

```
PL/M-86 COMPILER     i2Ø6ds.p86
                     Local Data

            $subtitle('Local Data')
               /*
                *  The need$reset array is used to determine if device needs to be
                *  reset after an error. Indexed by status.
                *
                *  TRUE = ØFFH
                *  FALSE = ØØØH
                */

 49   1        DECLARE
                   need$reset(24)      BYTE DATA(
                       FALSE,          /* Successful completion */
                       TRUE,           /* ID field miscompare */
                       FALSE,          /* Data field CRC error */
                       FALSE,          /* special for incorrect result$type */
                       TRUE,           /* Seek error */
                       FALSE,
                       FALSE,
                       FALSE,
                       FALSE,          /* Illegal Record Address */
                       FALSE,
                       FALSE,          /* ID Field CRC error */
                       TRUE,           /* Protocol error */
                       TRUE,           /* Illegal Cylinder Address */
                       FALSE,
                       FALSE,          /* Record not found */
                       FALSE,          /* Data Mark Missing */
                       FALSE,          /* Format Error */
                       FALSE,          /* Write Protected */
                       FALSE,
                       TRUE,           /* Write Error */
                       FALSE,
                       FALSE,
                       FALSE,
                       FALSE);         /* Drive Not Ready */
```

```
PL/M-86 COMPILER    i206ds.p86
                    Unit Status Array

              $subtitle('Unit Status Array')
                  /*
                   *  unit$status is used to set the unit status field in iors.
                   *  Indexed by status.
                   *
                   *  IO$UNCLASS =
                   *  IO$SOFT =
                   *  IO$HARD =
                   *  IO$OPRINT =
                   *  IO$WRPROT =
                   */

 50   1           DECLARE
                      unit$status(24)     BYTE DATA(
                          IO$UNCLASS,     /* Successful completion */
                          IO$SOFT,        /* ID field miscompare */
                          IO$SOFT,        /* Data field CRC error */
                          IO$HARD,        /* special for incorrect result$type */
                          IO$SOFT,        /* Seek error */
                          IO$UNCLASS,
                          IO$UNCLASS,
                          IO$UNCLASS,
                          IO$HARD,        /* Illegal Record Address */
                          IO$UNCLASS,
                          IO$SOFT,        /* ID Field CRC error */
                          IO$SOFT,        /* Protocol error */
                          IO$HARD,        /* Illegal Cylinder Address */
                          IO$UNCLASS,
                          IO$SOFT,        /* Record not found */
                          IO$SOFT,        /* Data Mark Missing */
                          IO$SOFT,        /* Format Error */
                          IO$WRPROT,      /* Write Protected */
                          IO$UNCLASS,
                          IO$SOFT,        /* Write Error */
                          IO$UNCLASS,
                          IO$UNCLASS,
                          IO$UNCLASS,
                          IO$OPRINT);     /* Drive Not Ready */

                  /*
                   *  drive$ready is used to find the drive ready bit
                   *  in the drive status.
                   */

 51   1           DECLARE
                      drive$ready(4)  BYTE DATA(020H,040H,010H,020H);
```

```
              $subtitle('i206$start')

                  /*
                   * i206$start
                   *      start procedure for the iSBC 206 controller.
                   *
                   * CALLING SEQUENCE:
                   *      CALL i206$start(iors$p, duib$p, ddata$p);
                   *
                   * INTERFACE VARIABLES:
                   *      iors$p       - I/O Request/Result segment pointer
                   *      duib$p       - pointer to Device-Unit Information Block
                   *      ddata$p      - device data segment pointer.
                   *
                   * CALLS:
                   *      io$206
                   *      format$206
                   *      send$206$iopb
                   *
                   * CALLED FROM:
                   *      radev via a reference in the device info table.
                   *
                   * ABSTRACT:
                   *      This is the device start procedure called by Random
                   *      Access Interface (radev).  The device is assumed to
                   *      have been initialized, any necessary resources
                   *      allocated and the interrupt task has already been
                   *      created.  All requests to any number of
                   *      iSBC 206 controller board's are funneled through this
                   *      procedure.  The reentrant nature of the procedure will
                   *      allow multiple invocations with only one copy of the
                   *      code.  The nature of the request is passed in as the
                   *      function code and sub-function fields of the IORS.
                   *      The function provides a simple method to DO CASE into
                   *      the required procedures.
                   */

 52   1           i206$start: PROCEDURE(iors$p, duib$p, ddata$p) PUBLIC REENTRANT;
 53   2               DECLARE
                          iors$p       POINTER,
                          duib$p       POINTER,
                          ddata$p      POINTER;
 54   2               DECLARE
                          iors         BASED iors$p IO$REQ$RES$SEG,
                          duib         BASED duib$p DEV$UNIT$INFO$BLOCK,
                          dinfo$p      POINTER,
                          dinfo        BASED dinfo$p I206$DEVICE$INFO,
                          uinfo$p      POINTER,
                          uinfo        BASED uinfo$p I206$UNIT$INFO,
                          ddata        BASED ddata$p IO$PARM$BLOCK$206,
                          base         WORD,
                          dummy        BYTE;

                      /*
                       * Initialize the local variables.
                       */
```

```
PL/M-86 COMPILER    i206ds.p86
                    i206$start

 55   2               dinfo$p = duib.device$info$p;
 56   2               base = dinfo.base;
 57   2               uinfo$p = duib.unit$info$p;

                      /*
                       * If we got called because of a restore operation
                       * then just return.
                       */
 58   2               IF (ddata.restore) THEN
 59   2                   RETURN;

 60   2           do$case$funct:
                      DO CASE iors.funct;
                          /*
                           * in the following calls the @ddata is literally
                           * iopb$p (i.e., the pointer to the iopb).
                           */

 61   3               case$read:
                          DO;
 62   4                       CALL io$206(base, iors$p, duib$p, @ddata);
 63   4                   END case$read;

 64   3               case$write:
                          DO;
 65   4                       CALL io$206(base, iors$p, duib$p, @ddata);
 66   4                   END case$write;

 67   3               case$seek:
                          DO;
 68   4                       CALL io$206(base, iors$p, duib$p, @ddata);
 69   4                   END case$seek;

 70   3               case$spec$funct:
                          DO;
 71   4                       IF iors.sub$funct = FS$FORMAT$TRACK THEN
 72   4                           CALL format$206(base, iors$p, duib$p, @ddata);
                              ELSE
 73   4                           DO;
                                      /*
                                       * Notifiy caller that this is an
                                       * Illegal Device Driver Request.
                                       */
 74   5                               iors.status = E$IDDR;
 75   5                               iors.actual = 0;
 76   5                               iors.done = TRUE;
 77   5                           END;
 78   4                   END case$spec$funct;

 79   3               case$attach$device:
                          DO;
 80   4                       dummy = (duib.dev$gran = 512);
 81   4                       IF ((input(sub$system$port) OR 073H) <> 0FBH) OR
                                  (((input(disk$config$port) AND
                                      SHL(010H,SHR(duib.unit,2))) <> 0) <> dummy) THEN
 82   4                           DO;
```

```
PL/M-86 COMPILER     i206ds.p86
                     i206$start


 83   5                           iors.status = E$IO;
 84   5                           iors.unit$status = IO$OPRINT;
 85   5                           iors.actual = 0;
 86   5                           iors.done = TRUE;
 87   5                           RETURN;
 88   5                       END;
 89   4                   ddata.inter = inter$on$mask;
 90   4                   ddata.instr = restore$op;
 91   4                   IF NOT send$206$iopb(base, @ddata) THEN
                              /*
                               * the board would not accept the iopb
                               * so...
                               */
 92   4                       DO;
 93   5                           iors.status = E$IO;
                                  /*
                                   * insert the result code into unit status
                                   * so the user has access to the code.
                                   * This will assist in debugging.
                                   */
 94   5                           iors.unit$status = IO$SOFT OR
                                              SHL(input(result$byte$port), 8);
 95   5                           iors.actual = 0;
 96   5                           iors.done = TRUE;
 97   5                       END;
 98   4               END case$attach$device;

 99   3           case$detach$device:
                      DO;
100   4                   iors.status = E$OK;
101   4                   iors.done = TRUE;
102   4               END case$detach$device;

103   3           case$open:
                      DO;
104   4                   iors.status = E$OK;
105   4                   iors.done = TRUE;
106   4               END case$open;

107   3           case$close:
                      DO;
108   4                   iors.status = E$OK;
109   4                   iors.done = TRUE;
110   4               END case$close;

111   3           END do$case$funct;

112   2       END i206$start;
```

```
PL/M-86 COMPILER    i206ds.p86
                    i206$interrupt

              $subtitle('i206$interrupt')

                  /*
                   * i206$interrupt
                   *      interrupt procedure for the iSBC 206 controller.
                   *
                   * CALLING SEQUENCE:
                   *      CALL i206$interrupt(iors$p, duib$p, ddata$p);
                   *
                   * INTERFACE VARIABLES:
                   *      iors$p      - I/O Request/Result segment pointer
                   *      duib$p      - pointer to Device-Unit Information Block
                   *      ddata$p     - device data segment pointer.
                   *
                   * CALLS:
                   *      i206$start
                   *      send$206$iopb
                   *      rq$send$message
                   *
                   * CALLED FROM:
                   *      radev via a reference in the device info table.
                   *
                   * ABSTRACT:
                   *      This procedure will handle the interrupts from the
                   *      iSBC 206 controller and will initiate any actions
                   *      necessary to recover from an error condition
                   *      (there are some conditions that are not recoverable).
                   */

 113   1          i206$interrupt: PROCEDURE(iors$p, duib$p, ddata$p)
                                                          PUBLIC REENTRANT;
 114   2              DECLARE
                          iors$p          POINTER,
                          duib$p          POINTER,
                          ddata$p         POINTER;
 115   2              DECLARE
                          iors        BASED iors$p IO$REQ$RES$SEG,
                          duib        BASED duib$p DEV$UNIT$INFO$BLOCK,
                          dinfo$p     POINTER,
                          dinfo       BASED dinfo$p I206$DEVICE$INFO,
                          ddata       BASED ddata$p IO$PARM$BLOCK$206,
                          temp        BYTE,
                          base        WORD,
                          spindle     WORD,
                          status      WORD;

                      /*
                       * Initialize the local variables.
                       */
 116   2              dinfo$p = duib.device$info$p;
 117   2              base = dinfo.base;
 118   2              spindle = shr(duib.unit, 2);        /* 4 units/spindle */

                      /*
                       * input from the result type port and
                       * mask out all the unused bits.
```

```
PL/M-86 COMPILER     i206ds.p86
                     i206$interrupt

                        */
119   2               IF (input(result$type$port) AND 3) = 0 THEN
120   2               done$int:
                          DO;
121   3                       status = input(result$byte$port);

122   3                       IF ddata.restore THEN
123   3                       did$restore:
                                  DO;
124   4                               ddata.restore = FALSE;
125   4                               ddata.status(spindle) = status;
126   4                               IF iors$p <> 0 THEN
                                      /*
                                       * There is a valid iors and we have
                                       * just returned from a restore operation
                                       * so, reinitiate the request.
                                       */
127   4                               restart:
                                          DO;
128   5                                       CALL i206$start(iors$p,
                                                              ddata$p,
                                                              duib$p);
129   5                                   END restart;
                                      /*
                                       * That is all we can do so ...
                                       */
130   4                               RETURN;
131   4                           END did$restore;

132   3                       ddata.status(spindle) = status;

133   3                       IF iors$p <> 0 THEN
134   3                       valid$iors:
                                  DO;
135   4                               IF status <> 0 THEN
136   4                               bad$status:
                                          DO;
137   5                                       iors.status = E$IO;
138   5                                       IF (status <= 010H) THEN
139   5                                           temp = status;
                                              ELSE
140   5                                           temp = shr(status, 4) + 00FH;
141   5                                       iors.unit$status = unit$status(temp)
                                                                   OR SHL(status,8);
142   5                                       iors.actual = 0;
143   5                                       iors.done = TRUE;
                                              /*
                                               * Index into the need$reset array
                                               * to determine the next course of
                                               * action.
                                               */
144   5                                       IF need$reset(ddata.status
                                                          (iors.unit / 4)) THEN
145   5                                       recalibrate:
                                                  DO;
                                                      /*
```

```
PL/M-86 COMPILER    i206ds.p86
                    i206$interrupt

                                                   * Note: must index drive
                                                   * select bits from iors.unit.
                                                   */
 146   6                                          ddata.inter = inter$on$mask;
 147   6                                          ddata.instr = restore$op;
 148   6                                          ddata.restore = send$206$iopb(
                                                                       dinfo.base,
                                                                       @ddata);
 149   6                                       END recalibrate;

 150   5                                 END bad$status;
 151   4                              ELSE ok$status:
                                         DO;
                                            /*
                                             * set actual = count as the status
                                             * indicated that the transfer worked.
                                             * This is done regardless of the
                                             * operation preformed.
                                             */
 152   5                                    iors.actual = iors.count;
 153   5                                    iors.done = TRUE;
 154   5                                 END ok$status;
 155   4                        END valid$iors;
 156   3                     END done$int;
 157   2                  ELSE status$int:
                             DO;
                                /*
                                 * Have arrived here because of an interrupt
                                 * initiated by the drive itself.
                                 * Could have been a drive ready or not ready
                                 * signal.
                                 */
 158   3                        temp = input(inter$stat$port);
 159   3                        DO spindle=0 TO 3;
 160   4                           IF (temp AND SHL(1, spindle)) <> 0 THEN
 161   4                              GOTO found$spindle;
 162   4                        END;
 163   3                     found$spindle:
                                spindle = SHL(spindle,2);
 164   3                        DO temp=spindle TO spindle+3;
 165   4                           IF ((input(result$byte$port) AND
                                      drive$ready(spindle)) = 0) THEN
                                      /*
                                       * let the user know the status
                                       * of the drive.
                                       */
 166   4                              CALL notify(temp, @ddata);
 167   4                        END;
 168   3                     END status$int;

 169   2               END i206$interrupt;
```

```
PL/M-86 COMPILER    i206ds.p86
                    i206$init

                $subtitle('i206$init')

                    /*
                     * i206$init
                     *      init procedure for the iSBC 206 controller.
                     *
                     * CALLING SEQUENCE:
                     *      CALL i206$init(duib$p, ddata$p, status$p);
                     *
                     * INTERFACE VARIABLES:
                     *      duib$p      - pointer to Device-Unit Information Block
                     *      ddata$p     - device data segment pointer.
                     *      status$p    - pointer to WORD indicating status of
                     *                        the operation.
                     *
                     * CALLS:
                     *      <none>
                     *
                     * CALLED FROM:
                     *      radev via a reference in the device info table.
                     *
                     * ABSTRACT:
                     *      initialize the hardware when called.
                     *      There is not much to do.
                     */

 170   1            i206$init: PROCEDURE(duib$p, ddata$p, status$p) PUBLIC REENTRANT;
 171   2                DECLARE
                            duib$p          POINTER,
                            ddata$p         POINTER,
                            status$p        POINTER;

 172   2                DECLARE
                            duib        BASED duib$p DEV$UNIT$INFO$BLOCK,
                            dinfo$p     POINTER,
                            dinfo       BASED dinfo$p I206$DEVICE$INFO,
                            ddata       BASED ddata$p IO$PARM$BLOCK$206,
                            status      BASED status$p WORD;
 173   2                DECLARE
                            i           WORD;

 174   2                dinfo$p = duib.device$info$p;

                        /*
                         * Reset iSBC 206 controller.
                         */

 175   2                output(reset$port) = 0;
 176   2                status = E$OK;

 177   2                ddata.restore = FALSE;

 178   2            END i206$init;

 179   1        END i206ds;


MODULE INFORMATION:

     CODE AREA SIZE     = 036AH     874D
     CONSTANT AREA SIZE = 0000H       0D
     VARIABLE AREA SIZE = 0000H       0D
     MAXIMUM STACK SIZE = 0046H      70D
     1101 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER      i206io.p86: iSBC 206 controller I/O Module
                      Module Header


SERIES-III PL/M-86 DEBUG X119 COMPILATION OF MODULE I206IO
OBJECT MODULE PLACED IN :F5:I206IO.OBJ
COMPILER INVOKED BY:  PLM86.86 :F5:I206IO.P86 COMPACT NOTYPE OPTIMIZE(3) ROM


                $title('i206io.p86: iSBC 206 controller I/O Module')
                $subtitle('Module Header')
   1            i206io: DO;

                /*
                 *  This module modifies the 206 parameter block
                 *      and passes the address of it to
                 *      the iSBC 206 controller.
                 *
                 *  CONTAINS:
                 *      io$206
                 *
                 * LANGUAGE DEPENDENCIES: COMPACT ROM OPTIMIZE(3)
                 */

                $include(:fl:icomon.lit)
            =   $save nolist
                $include(:fl:inutyp.lit)
            =   $save nolist
                $include(:fl:iiotyp.lit)
            =   $save nolist
                $include(:fl:iparam.lit)
            =   $save nolist
                $include(:fl:i206dv.lit)
            =   $save nolist
                $include(:fl:i206in.lit)
            =   $save nolist
                $include(:fl:iiors.lit)
            =   $save nolist
                $include(:fl:iduib.lit)
            =   $save nolist
                $include(:fl:itrsec.lit)
            =   $save nolist
                $include(:fl:iexcep.lit)
            =   $save nolist
                $include(:fl:iioexc.lit)
            =   $save nolist

                $include(:fl:i206dc.ext)
            =   $save nolist

                /*
                 * This module does the normal io (read, writes and seeks).
                 * Formatting a track is handled by i206fm.p86.
                 */

  31    1           DECLARE
                        i206$op$codes (*)    BYTE DATA(
                            READ$OP,
                            WRITE$OP,



PL/M-86 COMPILER      i206io.p86: iSBC 206 controller I/O Module
                      Module Header

                            SEEK$OP
                        );
```

```
PL/M-86 COMPILER     i206io.p86: iSBC 206 controller I/O Module
                     io$206: iSBC 206 controller I/O Module


                $subtitle('io$206: iSBC 206 controller I/O Module')
                    /*
                     * io$206
                     *      I/O module (read/write/seek)
                     *
                     * CALLING SEQUENCE:
                     *      CALL io$206 (base, iors$p, duib$p, iopb$p);
                     *
                     * INTERFACE VARIABLES:
                     *      base         - base address of the board.
                     *      iors$p       - I/O Request/Result segment pointer
                     *      duib$p       - pointer to Device-Unit Information Block
                     *      iopb$p       - pointer to I/O parameter block.
                     *
                     * INTERNAL VARIABLES:
                     *      iors         - I/O Request/Result Structure.
                     *      ts           - DWORD containing track and sector info.
                     *      ts$o         - overlay of ts to allow access through
                     *                       PL/M-86.
                     *      duib         - Device Unit Information Block Structure.
                     *      iopb         - I/O parameter block for the
                     *                       iSBC 206 controller.
                     *      platter      - local var to prevent multiple computations.
                     *      spindle      - as above.
                     *      surface      - as above.
                     *
                     * CALLS:
                     *      send$206$iopb(base, @iopb)
                     *
                     * ABSTRACT:
                     *      All io functions (except format) are handled by this
                     *      module.
                     */
  32   1            io$206: PROCEDURE (base, iors$p, duib$p, iopb$p) REENTRANT PUBLIC;
  33   2                DECLARE
                            base         WORD,
                            iors$p       POINTER,
                            duib$p       POINTER,
                            iopb$p       POINTER;
  34   2                DECLARE
                            iors         BASED iors$p IO$REQ$RES$SEG,
                            ts           DWORD,
                            ts$o         TRACK$SECTOR$STRUCT AT(@ts),
                            duib         BASED duib$p DEV$UNIT$INFO$BLOCK,
                            iopb         BASED iopb$p IO$PARM$BLOCK$206,
                            platter      BYTE,
                            spindle      BYTE,
                            surface      BYTE;

                /*
                 * Initialize local variables:
                 *  ts <-- track and sector info from iors.dev$loc.
                 *  platter <-- from iors.unit.
                 *  spindle <-- from iors.unit.
                 *  surface <-- from high bit in track field.
                 */
```

```
PL/M-86 COMPILER     i206io.p86: iSBC 206 controller I/O Module
                     io$206: iSBC 206 controller I/O Module


 35   2           ts = iors.dev$loc;

 36   2           spindle = shr(iors.unit, 2);       /* 4 units/spindle */
 37   2           platter = iors.unit AND 003H;      /* (as above ) */
 38   2           surface = ts$o.track AND 00001H;   /* select surface */

                  /*
                   * Fill out the iopb for the iSBC 206 controller.
                   */
 39   2           iopb.inter = INTER$ON$MASK;        /* we use interrupts */
 40   2           iopb.cyl$add = shr(ts$o.track, 1); /* track/2 = cylinder */

                  /*
                   * Note that the iopb.instr field is used by
                   * the iSBC 206 controller to determine which
                   * drive/platter/surface combination to access
                   * AND the op code determines
                   * how that combination is to be accessed.
                   */
 41   2           iopb.instr = i206$op$codes(iors.funct) OR
                               shl(spindle, 4) OR
                               shl(platter, 6) OR
                               shl(surface, 3);

                  /*
                   * note: the controller only supports 512
                   * or 128 byte sectors so no checking is done.
                   */

                  /* divide by sectors size */
 42   2           iopb.r$count = iors.count / duib.dev$gran;

                  /*
                   * sectors come in based on 0 and the controller
                   * will only understand sectors starting at 1.
                   */

                  /* (cyl AND 0100H) / 2 */
 43   2           iopb.rec$add = (ts$o.sector + 1) OR
                                  shr(ts$o.track AND 0200H, 2);

 44   2           iopb.buff$p = iors.buff$p;

 45   2           IF NOT send$206$iopb(base, @iopb) THEN
                      /*
                       * the board did not accept the iopb so...
                       */
 46   2               DO;
 47   3                   iors.status = IO$SOFT;
 48   3                   iors.actual = 0;
 49   3                   iors.done = TRUE;
 50   3               END;

 51   2       END io$206;

 52   1   END i206io;

PL/M-86 COMPILER     i206io.p86: iSBC 206 controller I/O Module
                     io$206: iSBC 206 controller I/O Module


MODULE INFORMATION:

     CODE AREA SIZE     = 00DBH    219D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0022H     34D
     615 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER     i2Ø6dc: iSBC 2Ø6 controller parameter handler
                     Module Header

SERIES-III PL/M-86 DEBUG X119 COMPILATION OF MODULE I2Ø6DC
OBJECT MODULE PLACED IN :F5:I2Ø6DC.OBJ
COMPILER INVOKED BY:  PLM86.86 :F5:I2Ø6DC.P86 COMPACT NOTYPE OPTIMIZE(3) ROM


                 $title('i2Ø6dc: iSBC 2Ø6 controller parameter handler')
                 $subtitle('Module Header')
   1             i2Ø6dc: DO;

                 /*
                  *  i2Ø6dc.p86
                  *
                  *  CONTAINS:
                  *      send$2Ø6$iopb
                  *
                  * LANGUAGE DEPENDENCIES: COMPACT ROM OPTIMIZE(3)
                  */

                 $include(:f1:icomon.lit)
            =    $save nolist
                 $include(:f1:inutyp.lit)
            =    $save nolist
                 $include(:f1:i2Ø6dv.lit)
            =    $save nolist
```

```
PL/M-86 COMPILER    i206dc: iSBC 206 controller parameter handler
                    Send 206 I/O Parameter Block

              $subtitle('Send 206 I/O Parameter Block')
                  /*
                   * send$206$iopb
                   *      send the iSBC 206 controller the address of the parameter block
                   *
                   * CALLING SEQUENCE:
                   *      CALL send$206$iopb (base, iopb$p);
                   *
                   * INTERFACE VARIABLES:
                   *      base         - base address of board.
                   *      iopb$p       - I/O parameter block pointer
                   *
                   * INTERNAL VARIALBLES:
                   *      iopb$p$o     - overlay for the pointer.
                   *      iopb         - I/O parameter block structure.
                   *      drive        - local var to reduce computations.
                   *
                   * CALLS:
                   *      <none>
                   *
                   * ABSTRACT:
                   *      outputs the iopb to the iSBC 206 controller.
                   */
  8    1          send$206$iopb: PROCEDURE (base, iopb$p) BOOLEAN REENTRANT PUBLIC;
  9    2              DECLARE
                          base         WORD,
                          iopb$p       POINTER;
 10    2              DECLARE
                          iopb$p$o     P$OVERLAY AT(@iopb$p),
                          iopb         BASED iopb$p IO$PARM$BLOCK$206,
                          drive        BYTE;

                      /*
                       * Extract the drive unit from the instruction.
                       */
 11    2              drive = shr(iopb.instr AND 030H, 4);
 12    2              drive = shl(01H,drive);

                      /*
                       * Check to see if the drive is busy.
                       */
 13    2              IF (input(controller$stat)) <> (COMMAND$BUSY OR drive) THEN
 14    2                  DO;
 15    3                      output (lo$off$port) = low (iopb$p$o.offset);

                              /*
                               * Check to see if the drive is busy AGAIN.
                               */
 16    3                      IF (input(controller$stat) AND COMMAND$BUSY) = 0 THEN
 17    3                          DO;
                                      /*
                                       * made it to here so
                                       * output rest of iopb address.
                                       */
 18    4                              output (lo$seg$port) = low (iopb$p$o.base);
 19    4                              output (hi$seg$port) = high (iopb$p$o.base);
```

```
PL/M-86 COMPILER     i206dc: iSBC 206 controller parameter handler
                     Send 206 I/O Parameter Block

 20   4                                output (hi$off$port) = high (iopb$p$o.offset);

 21   4                                RETURN(TRUE);
 22   4                            END;
 23   3                    END;

                       /*
                        * If we got here then something blew up.
                        * So inform the caller that we could not process the iopb.
                        */
 24   2                RETURN (FALSE);

 25   2            END send$206$iopb;

 26   1        END i206dc;


MODULE INFORMATION:

     CODE AREA SIZE     = 0066H    102D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 000CH     12D
     216 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER     i206fm.p86
                     Module Header

SERIES-III PL/M-86 DEBUG X119 COMPILATION OF MODULE I206FM
OBJECT MODULE PLACED IN :F5:I206FM.OBJ
COMPILER INVOKED BY:  PLM86.86 :F5:I206FM.P86 COMPACT NOTYPE OPTIMIZE(3) ROM


                 $title('i206fm.p86')
                 $subtitle('Module Header')

                 /*
                  *  i206fm.p86
                  *
                  * CONTAINS:
                  *     format$206
                  *     build206$$fmt$table
                  *
                  * LANGUAGE DEPENDENCIES: COMPACT ROM OPTIMIZE(3)
                  */

   1             i206fm: DO;

                 $include(:fl:icomon.lit)
             =   $save nolist
                 $include(:fl:inutyp.lit)
             =   $save nolist
                 $include(:fl:iiotyp.lit)
             =   $save nolist
                 $include(:fl:iparam.lit)
             =   $save nolist
                 $include(:fl:i206dv.lit)
             =   $save nolist
                 $include(:fl:i206in.lit)
             =   $save nolist
                 $include(:fl:iradsf.lit)
             =   $save nolist
                 $include(:fl:iiors.lit)
             =   $save nolist
                 $include(:fl:iduib.lit)
             =   $save nolist
                 $include(:fl:itrsec.lit)
             =   $save nolist
                 $include(:fl:iexcep.lit)
             =   $save nolist
                 $include(:fl:iioexc.lit)
             =   $save nolist

                 $include(:fl:i206dc.ext)
             =   $save nolist
```

```
PL/M-86 COMPILER     i206fm.p86
                     format$206: Format track procedure


               $subtitle('format$206: Format track procedure')
                   /*
                    * format$206
                    *      format a track on the iSBC 206 controller.
                    *
                    * CALLING SEQUENCE:
                    *      CALL format$206 (base, iors$p, duib$p, iopb$p);
                    *
                    * INTERFACE VARIABLES:
                    *      base         - base address of board.
                    *      iors$p       - I/O Request/Result segment pointer
                    *      duib$p       - pointer to Device-Unit Information Block
                    *      iopb$p       - I/O parameter block pointer.
                    *
                    * CALLS:
                    *      build$206$fmt$table
                    *      send$206$iopb
                    *
                    * CALLED FROM:
                    *      i206$start
                    *
                    * ABSTRACT:
                    *      this procedure will format a single track on the disk.
                    *      It will not format the other side of the cylinder.
                    */
 34   1            format$206: PROCEDURE (base, iors$p, duib$p, iopb$p)
                                                             REENTRANT PUBLIC;
 35   2                DECLARE
                           base            WORD,
                           iors$p          POINTER,
                           duib$p          POINTER,
                           iopb$p          POINTER;
 36   2                DECLARE
                           iors        BASED iors$p IO$REQ$RES$SEG,
                           format$info$p   POINTER,
                           format$info BASED format$info$p FORMAT$INFO$STRUCT,
                           duib        BASED duib$p DEV$UNIT$INFO$BLOCK,
                           iopb        BASED iopb$p IO$PARM$BLOCK$206,
                           platter             BYTE,
                           spindle             BYTE,
                           surface             BYTE,
                           max$sectors         BYTE;

                       /*
                        * initialize local variables.
                        */
 37   2                format$info$p = iors.aux$p;
 38   2                IF format$info.track$num > i205$TRACK$MAX THEN
 39   2                    DO;
                               /*
                                * Let's leave now since we cannot
                                * access any tracks.
                                */
 40   3                        iors.status = E$SPACE;
 41   3                        iors.actual = 0;
 42   3                        iors.done = TRUE;
```

```
  43   3                   RETURN;
  44   3               END;

              /*
               * use local variables to eliminate later confusion.
               */
  45   2      spindle = shr(iors.unit, 2);        /* 4 units/spindle */
  46   2      platter = iors.unit AND 003H;       /* (as above ) */
  47   2      surface = format$info.track$num AND
                              00001H;             /* select surface */

              /*
               * fill out the IOPB for the io$206.
               */
  48   2      iopb.inter = INTER$ON$MASK OR FORMAT$TRACK$ON;

              /* track/2 = cylinder */
  49   2      iopb.cyl$add = shr(format$info.track$num, 1);

              /* set bit if over 256 cylinders */
  50   2      iopb.rec$add = shr(format$info.track$num AND 0200H, 2);
  51   2      iopb.instr = format$op         OR
                           shl(spindle, 4) OR
                           shl(platter, 6) OR
                           shl(surface, 3);

  52   2      iopb.buff$p = @iopb.format$table;

  53   2      IF duib.dev$gran = 128 THEN
  54   2          max$sectors = 36;
              ELSE
                  /*
                   * if not 128 then MUST be 512 byte sectors
                   */
  55   2          max$sectors = 12;

              /*
               * the device controller expects a table built containing
               * the information on what the track should look like.
               * so build it using the local variable.
               */
  56   2      CALL build$206$fmt$table(@iopb.format$table,
                                        format$info.track$num,
                                        format$info.track$interleave,
                                        format$info.track$skew,
                                        format$info.fill$char,
                                        max$sectors);

  57   2      IF NOT send$206$iopb(base, @iopb) THEN
                  /*
                   * the board did not accept the iopb so...
                   */
  58   2          DO;
  59   3              iors.status = IO$SOFT;
  60   3              iors.actual = 0;
  61   3              iors.done = TRUE;
```



```
  62   3          END;

  63   2  END format$206;
```

```
              $eject
                  /*
                   * build$2Ø6$fmt$table
                   *      fill out format table
                   *
                   * CALLING SEQUENCE:
                   *      CALL build$2Ø6$fmt$table(buf$p,
                   *                              track,
                   *                              int$fact,
                   *                              skew,
                   *                              fill$char,
                   *                              max$sectors);
                   *
                   * INTERFACE VARIABLES:
                   *      buf$p       - address of format table.
                   *      track       - track to be formatted.
                   *      int$fact    - interleave factor.
                   *      skew        - squew from physical  sector one.
                   *      fill$char   - used to fill sectors.
                   *      max$sectors - maximum number of sectors
                   *
                   * CALLS:
                   *      <none>
                   *
                   * No error checking on skew, int$fact parameters;
                   *  if nonsense, the algorithm completes & formats
                   *  the track in a strange manner.
                   */
 64   1           build$2Ø6$fmt$table: PROCEDURE(buf$p, track, int$fact, skew,
                                        fill$char,max$sectors) REENTRANT;
 65   2               DECLARE
                          buf$p        POINTER,
                          track        WORD,
                          int$fact     BYTE,
                          skew         BYTE,
                          fill$char    BYTE,
                          max$sectors  BYTE;
 66   2               DECLARE
                          s            BYTE,
                          i            BYTE;
 67   2               DECLARE
                          fmt$tab BASED buf$p (36) STRUCTURE(
                              record$address  BYTE,
                              fill$char       BYTE);

                      /*
                       * fill out the format table with ØFFH,
                       * this will be used to indicate when
                       * all the record addresses are filled in.
                       */
 68   2               DO i = Ø TO (max$sectors - 1);
 69   3                   fmt$tab(i).record$address = ØFFH;
 7Ø   3                   fmt$tab(i).fill$char = fill$char;
 71   3               END;
```

```
PL/M-86 COMPILER     i206fm.p86
                     format$206: Format track procedure

 72   2                  s = skew MOD max$sectors;

 73   2                  DO i = 1 TO max$sectors;

 74   3                      DO WHILE fmt$tab(s).record$address <> 0FFH;
 75   4                          s = (s + 1) MOD max$sectors;
 76   4                      END;

 77   3                      fmt$tab(s).record$address = i;
 78   3                      s = (s + int$fact) MOD max$sectors;

 79   3                  END;

 80   2              END build$206$fmt$table;

 81   1          END i206fm;


MODULE INFORMATION:

     CODE AREA SIZE     = 0195H     405D
     CONSTANT AREA SIZE = 0000H       0D
     VARIABLE AREA SIZE = 0000H       0D
     MAXIMUM STACK SIZE = 0028H      40D
     717 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

INDEX

Underscored entries are primary references.

REQUEST FOR READER'S COMMENTS

Intel Corporation attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. ____________

NAME ______________________________ DATE ____________

TITLE ______________________________

COMPANY NAME/DEPARTMENT ______________________________

ADDRESS ______________________________

CITY ____________________ STATE ________ ZIP CODE ________

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 N.E. Elam Young Pkwy.
Hillsboro, Oregon 97123

O.M.S. Technical Publications

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) 987-8080

Printed in U.S.A.