GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86™ I/O SYSTEM

Manual Order Number: 142926-002

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

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

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

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

BXP	Int$_{el}$	Megachassis
CREDIT	Intelevision	Micromap
i	Intellec	Multibus
ICE	iRMX	Multimodule
iCS	iSBC	PROMPT
im	iSBX	Promware
Insite	Library Manager	RMX/80
Intel	MCS	System 2000
		UPI
		μScope

and the combination of ICE, iCS, iRMX, iSBC, iSBX, MCS, or RMX and a numerical suffix.

A367/681/5K DL

PREFACE

The I/O System is the part of the iRMX 86 Operating System that provides you with the capability to access files on peripheral devices. It 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 System. It illustrates the basic concepts of device drivers in an iRMX 86 environment and describes the different types of device drivers (common, random access, and custom).

READER LEVEL

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

- 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.
- 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.

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	143318
iRMX 86 System Programmer's Reference Manual	142721
iRMX 86 Configuration Guide	9803126
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

CONTENTS (continued)

FIGURES

CHAPTER 1. INTRODUCTION

The I/O System is 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 206 device driver provides the facilities that enable an iSBC 206 disk 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 System is 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.

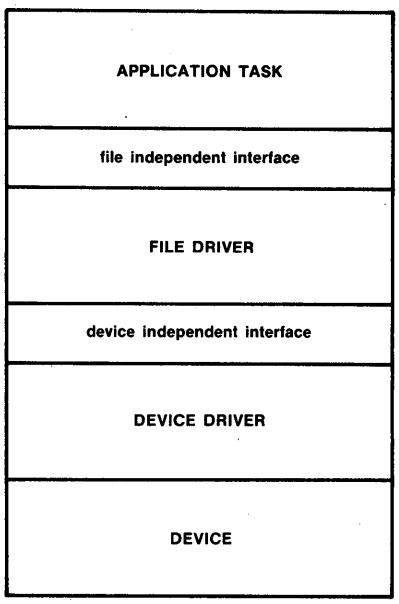


Figure 1-1. Communication Levels

The I/O System provides a standard interface between file drivers and device drivers. To a file driver, any 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 only have to 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 of the devices. Figure 1-2 contains a simplified drawing of three I/O devices and their device, unit, and device-unit numbers.

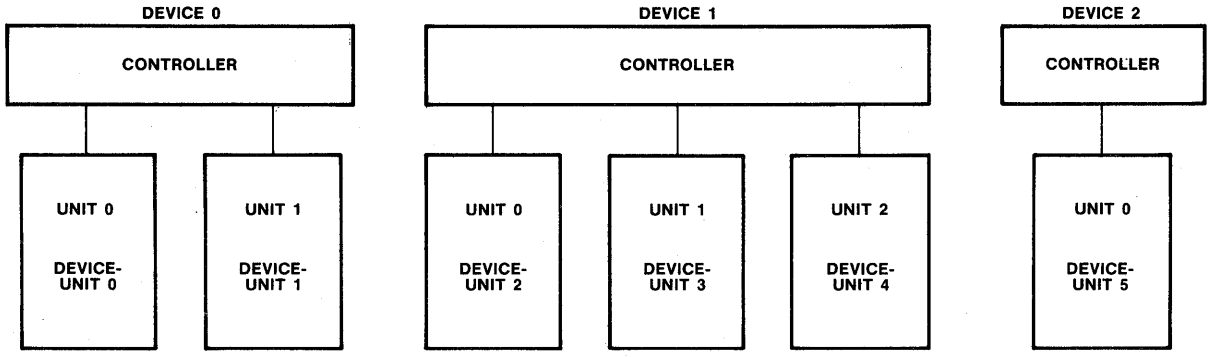


Figure 1-2. Device Numbering

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 206 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 include:

- 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 I/O port read and write commands in order to cause the device to perform the requested operation.

COMPONENTS OF A DEVICE DRIVER

At its highest level, a device driver consists of four procedures which are called directly by the I/O System. These procedures can be identified according to purpose, as follows:

- Initialize I/O
- Finish I/O
- Queue I/O
- Cancel I/O

When a task makes an I/O System call to manipulate a device, the I/O System ultimately calls one or more of these procedures, which operate in conjunction with an interrupt handler to coordinate the actual I/O transfers. This section provides a general description of each of these procedures and the interrupt handler.

INITIALIZE I/O

This procedure creates all of the iRMX 86 objects needed by the device driver. It typically creates an interrupt task and a segment to store data local to the device. It also performs device initialization, if any is necessary. The I/O System calls this procedure when an application task makes an RQAPHYSICAL$ATTACH$DEVICE system call to attach a unit of a device and there are no units of the device currently attached. Refer to the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL for a description of the system call.

FINISH I/O

The I/O System calls this procedure when an application task makes an RQAPHYSICAL$DETACH$DEVICE system call to detach a unit of a device and that unit is the only unit of the device currently attached (refer to the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL for a description of the system call). The Finish I/O procedure performs any necessary final processing on the device and deletes all of the objects created by the Initialize I/O procedure, including the interrupt task and the segment containing the data local to the device.

QUEUE I/O

This procedure places I/O requests on a queue, so that they can be processed when the appropriate unit becomes available. If the device is not busy, the Queue I/O procedure starts the request after placing it on the queue.

CANCEL I/O

This procedure cancels a previously queued I/O request. This function is useful, for example, when a request can take an unacceptable amount of time to process (such as keyboard input from a terminal).

INTERRUPT HANDLERS AND INTERRUPT TASKS

After a device finishes processing an I/O 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 signals 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 (refer to the iRMX 86 NUCLEUS REFERENCE MANUAL for a description of this), 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. The interrupt task is normally created by the Initialize I/O procedure.

Figure 1-3 shows the interaction of 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.

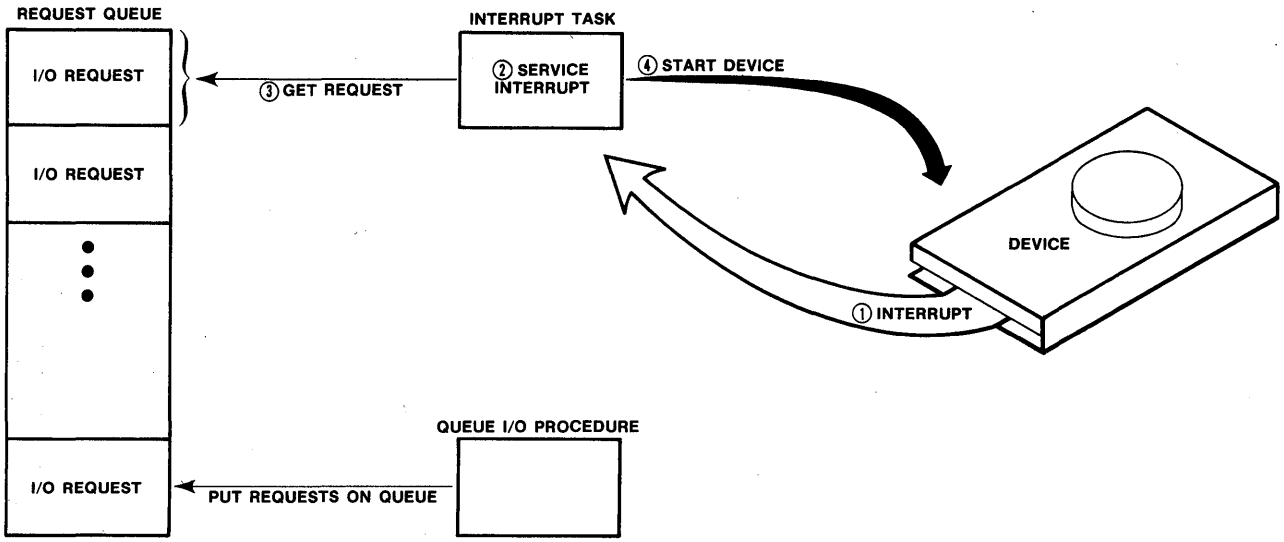


Figure 1-3. Interrupt Task Interaction

CALLING THE DEVICE DRIVER PROCEDURES

The I/O System calls each one of the four device driver procedures in response to specific conditions. Figure B-1 is flow chart that illustrates the conditions under which three of the four procedures are called. You can unfold this figure in order to follow along with the discussion of it. The following numbered paragraphs discuss the portions of Figure B-1 labeled with corresponding circled numbers.

1. In order start I/O processing, the user must make an I/O request. This can be done by making a variety of system calls. However, the first I/O request to each device-unit must be the RQAPHYSICAL$ATTACH$DEVICE system call.

2. If the request results from an RQAPHYSICAL$ATTACH$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 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 objects used by the device driver routines.

The 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.

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 can be in any form or 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 common to random access or common devices. If your device fits into the random access or common 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.

COMMON DEVICES

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

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

- 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.

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 3 of this manual describes common device drivers in detail.

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:

- The device supports random access seek.
- Only one interrupt level is needed to service the device.
- I/O requests must be broken up into blocks of a specific length.

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 4 describes random access device drivers in detail.

CUSTOM DEVICES

If your device fits neither the common nor the random access category, you must write the entire driver for that device. Custom device drivers are discussed in Chapter 5.

CHAPTER 2. DEVICE DRIVER INTERFACES

Since a device driver is a group of software routines which manages a device at a basic level, it must transform general instructions that it receives 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 descusses 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, the I/O System can call a device driver. Actually, the DUIB is the interface between the I/O System and the device itself. It is a part of a table that contains (or points to) all of the information that the I/O System knows about a particular unit of a device. 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.

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. When you attach a device using the RQAPHYSICAL$ATTACH$DEVICE system call, you must specify this name as a parameter. 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. Bits are numbered from right to left, starting with 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. 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. Bits are numbered from right to left, starting with 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; for example, the RQSGET$FILE$STATUS system call (refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL) uses this field. 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	reserved
1	0 = single density; 1 = double density
2	0 = single sided; 1 = double sided
3-7	reserved

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. For example, if you format an iRMX 86 disk and set its granularity to 128, you should also set the device granularity to 128.

LOWDEVSIZE HIGHDEVSIZE	WORD 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 this unit from the rest of the units of the device.
DEV$UNIT	WORD specifying the device-unit number. This number distinguishes the device-unit from the rest of the units in the entire hardware system. Device drivers can ignore this field.
INIT$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$P	POINTER to a structure which contains additional information about the device. The command and random access device drivers require device information structures of a particular format. These structures are described in Chapters 3 and 4. If you are writing a custom driver, you can place information in this structure depending on the needs of your driver. Specify a zero for this parameter if the associated device driver does not use this field.
UNIT$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. 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 the random access device driver support routines 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, these procedures 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, you must first attach the device. You do this 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). One of the things that this system call does is to associate a particular device-unit with a DUIB. This is done when you specify the DUIB name as the dev$name parameter of RQ$A$PHYSICAL$ATTACH$DEVICE.

When you do this, the I/O System assumes that the device-unit identified by the device number, unit number, and device-unit number fields 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 assumes that the characteristics of that unit, including which device driver procedures to call in order to actually process the I/O request, are those 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 name parameters, you can choose which DUIB to associate with a device-unit that you are attaching by specifying different dev$name parameters on the RQAPHYSICAL$ATTACH$DEVICE system call.

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 call the RQAPHYSICAL$DETACH$DEVICE system call to detach the device, and attach it again with RQAPHYSICAL$ATTACH$DEVICE using the other DUIB name.

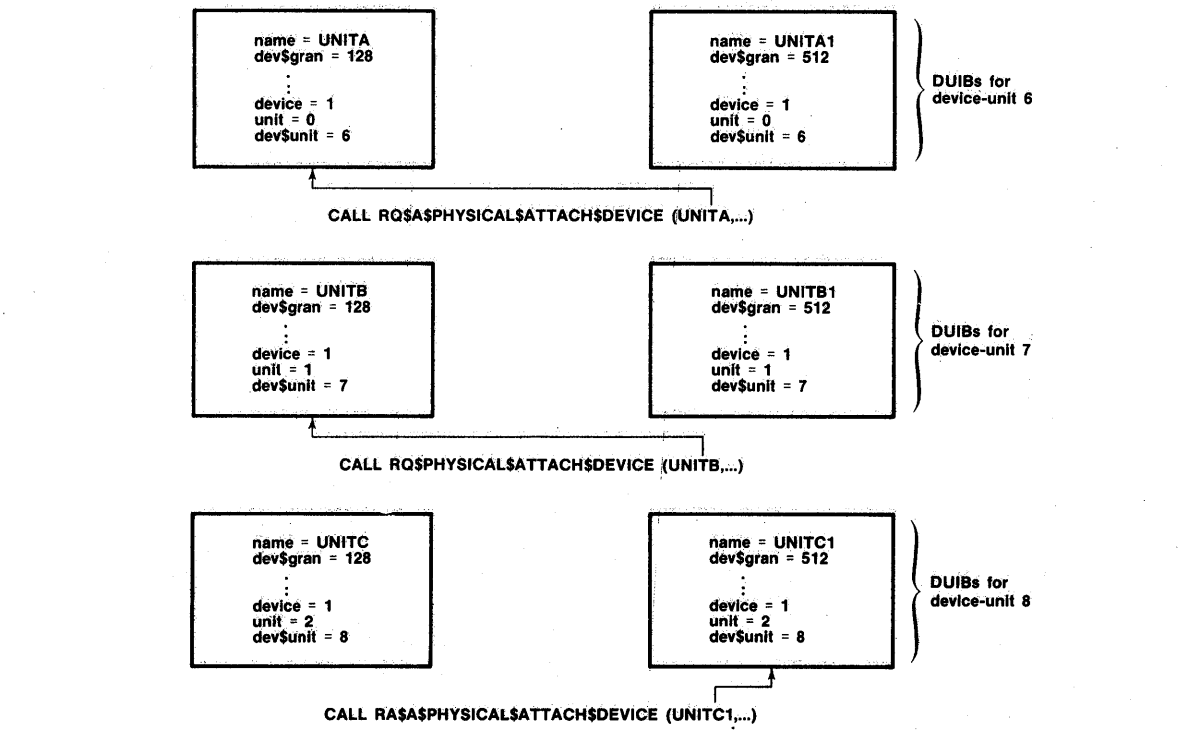


Figure 2-1. Attaching Devices

Creating DUIBs

Before the system starts running, you must create all of the DUIBs that you will ever need. You cannot create additional ones at run time. 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. A device driver processes I/O for a device as a whole.

- If you are using a common or random access device driver, you must create a device information table for each device and a unit information table for each type of unit (see Chapter 4). Place pointers to these tables in the device$info$p and unit$info$p fields of the appropriate DUIBs. If your custom device drivers require these tables, you must create them at this time.

I/O REQUEST/RESULT SEGMENT (IORS)

An I/O request/result segment (IORS) is an iRMX 86 object that the I/O System creates when a user requests an I/O operation. It is a data structure that contains information about the request and about the unit on which the operation is to be performed. The I/O System passes the token for the IORS to the appropriate device driver which then must process 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 indicated in the IORS.

IORS Structure

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.

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 and iSBC 206 drivers place the result byte in the high eight bits of this field. Refer to the hardware reference manuals for the iSBC 204 and iSBC 206 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 a random access device driver, other drivers should avoid using these codes for other functions.

LOWDEVLOC
HIGHDEVLOC — WORDs 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$WRITE operation, this is the address on the device where the write begins. If a random access device driver is used and the track$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 buffer in iRMX 86 memory 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 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:

```
DELCARE   FORMAT$TRACK   STRUCTURE(
    TRACK$NUMBER      WORD,
    INTERLEAVE        WORD,
    TRACK$OFFSET      WORD,
    FILL$CHAR         WORDK,
```

The other 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 a token for the response mailbox. Upon completion of the I/O request, the device driver must send the IORS to this response mailbox.

DONE — BYTE that the device driver can set to TRUE (0FFH) or FALSE (00H) 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.

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 random access and common device drivers form this queue by creating a doubly linked list.

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 random access and common 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 2-2.

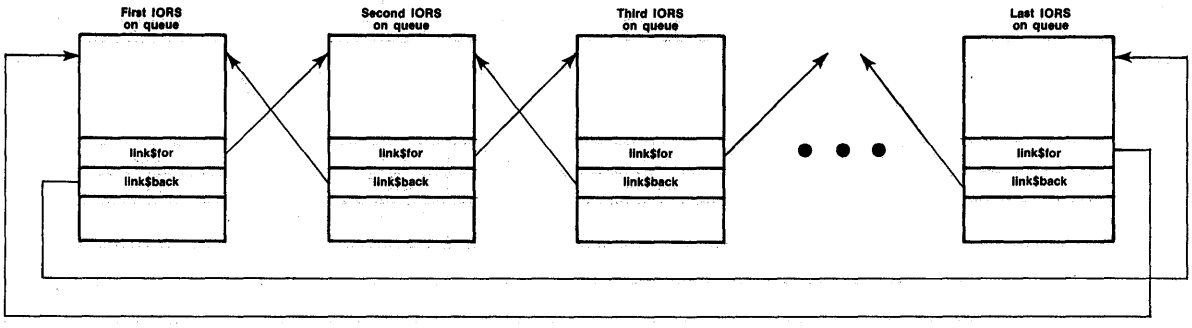


Figure 2-2. Request Queue

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

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

Device Data Object

The common and random access device drivers are set up so that all data which is local to a device is maintained in an iRMX 86 segment. This segment is called a device data object. The Initialize I/O procedure creates this device data object and the other procedures of the driver access and update information in it as needed. Two purposes are served by keeping the device-local data in a segment.

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 a token for the segment back to the I/O System, which in turn gives the token to the other procedures of the driver. They can then place information relevant to the device as a whole into the segment. The identity of the first IORS on the request queue is maintained in this segment, as well as the attachment status of the individual units and a token for 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 creates a segment for the device data. However, the segments it creates are different. Only the incarnations of the routines that service units of a particular device are able to access the device data object for that device.

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

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. WRITING A COMMON DEVICE DRIVER

Common devices are simple devices, such as line printers, that do not require a great deal of manipulation by a device driver. They are generally sequential devices that do not have sector or track data formatting requirements. The I/O System supplies a number of the support procedures necessary to implement a device driver for these common devices. These support procedures handle the creation and deletion of objects needed by the driver, interrupts, and queuing of requests. By using the I/O System-supplied support procedures, you can save the time and expense of creating your own procedures to perform the same functions. All you need to do is supply the device-dependent routines which initialize the device, send data to the device, respond to interrupts, and close down the device.

The remainder of this chapter contains the requirements for using the common device driver support routines supplied by the I/O System, a list of these routines, and the requirements of the user-supplied routines. A discussion of the common driver routines supplied by the I/O System is contained in Appendix A.

REQUIREMENTS FOR USING THE COMMON DEVICE DRIVER SUPPORT ROUTINES

Any device can make use of the common device driver support routines, as long as the device can conform to the following conditions:

- Data written or read by this device does not need to conform to sector or track boundaries. The common driver support routines do not guarantee that data starts at sector or track boundaries.
- A first in/first out queuing mechanism is sufficient for making I/O requests.
- Only one interrupt level is necessary in order to service the device. The common device driver support routines do not support separate input and output interrupt levels, for instance.

I/O SYSTEM-SUPPLIED ROUTINES

The I/O System supplies each of the highest-level common device driver procedures. These procedures are:

INIT$IO	Initialize I/O procedure
FINISH$IO	Finish I/O procedure
QUEUE$IO	Queue I/O procedure
CANCEL$IO	Cancel I/O procedure

Use these names when creating DUIBs for devices that make use of the common device driver.

Each one of these procedures calls additional procedures, some of which are supplied by the I/O System and some of which must be supplied by you. The following sections discuss the routines that you must supply. Appendix A contains a further discussion of the I/O System-supplied procedures.

USER-SUPPLIED ROUTINES

The routines provided by the I/O System for the common device driver make up a majority of the routines in a common device driver. However, these routines make calls to device-dependent routines that you must supply. These device-dependent routines are:

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.

You can write these routines in either PL/M-86 or assembly language. However, you must adhere to the following guidelines:

- If you use PL/M-86, you must define your routines as reentrant 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 case. 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.

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 in a common device information table. Each DUIB contains a pointer to a device information table (refer to Chapter 2). This table contains not only the addresses of the user-supplied routines, but also other device-specific information.

DUIBs which correspond to units of the same device should point to the same device information table.

DEVICE INFORMATION TABLE FOR COMMON DEVICES

You must place the device-specific information for each common device in a device information table, as follows:

```
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.

PRIORITY	BYTE specifying the initial priority of the interrupt task. The actual priority of the interrupt task may change due to the fact that the 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 object. This figure should not include the amount needed by the I/O System-supplied procedures, but 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 is described later in this chapter.
DEVICE$FINISH	WORD specifying the start address of a user-written device finish procedure. The format of this procedure is described later in this chapter.
DEVICE$START	WORD specifying the start address of a user-written device start procedure. The format of this procedure is described later in this chapter.
DEVICE$STOP	WORD specifying the start address of a user-written device stop procedure. The format of this procedure is described later in this chapter.
DEVICE$INTERRUPT	WORD specifying the start address of a user-written device interrupt procedure. The format of this procedure is described later in this chapter.

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

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$p	POINTER to the user portion of the device's data object. You must specify the size of this portion in the device information table for this device. The device initialization procedure can use this data area for whatever purposes it chooses. Possible uses for this data area include local flags and buffer areas.
status$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 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 object. The device finish procedure should obtain, from the data object, tokens for any objects that other user-written procedures may have created, and delete these objects.

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 and is normally used when the default initialization procedure DEFAULT$INIT is used.

DEVICE START PROCEDURE

Both QUEUE$IO and the interrupt task make calls to the device start procedure in order to start an I/O function. QUEUE$IO calls this procedure on receiving an I/O request when the request queue is empty. The 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 object. 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$STATUS field to a nonzero value. The lower four bits of the field should be set as indicated in the "IORS Structure" section of Chapter 2. The remaining bits of the field can be set to any value (for example, the iSBC 204 and 206 device drivers return the device's result byte in the remainder of this field). The device start procedure must also set the IORS.DONE field to TRUE, indicating that the request is done because of the error. If the function completes without an error, the device start procedure must set the IORS.STATUS field to indicate an E$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 common device driver support routines will wait until the device sends an interrupt and the device interrupt procedure sets 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 object. 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

The 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 object. 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 object, 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 4. WRITING A RANDOM ACCESS DEVICE DRIVER

Random access devices are I/O devices, such as disk drives, bubble memories, and other such devices from which data can be read, onto which data can be written, and on which you can seek to any memory location in order to perform these operations. The I/O System supplies a number of the support procedures that are necessary to implement a random access device driver. These support procedures provide the same features and procedure interfaces as the I/O System-supplied common device driver support procedures supply: the creation and deletion of objects needed by the driver routines, an interrupt handler, and queuing of requests. Moreover, the random access driver procedures supplied by the I/O System guarantee that I/O requests conform to sector and track requirements of the device.

In order to use the random access device driver support procedures, you must create the device-dependent routines which initialize the device, send data to the device, respond to an interrupt, and close down the device. You must combine these with the I/O System-supplied support procedures.

The remainder of this chapter contains the requirements for using the random access device driver routines supplied by the I/O System, a list of these routines, and the requirements for the user-supplied routines.

REQUIREMENTS FOR USING THE RANDOM ACCESS DEVICE DRIVER SUPPORT ROUTINES

Devices can make use of the random access device driver support routines if they conform to the following conditions:

- The device must support random access seek operations.
- A first in/first out queuing mechanism is sufficient for making I/O requests. The random access device driver support routines maintain the I/O request queue as a FIFO queue.
- Only one interrupt level is necessary to support the device. The random access device driver routines do not support separate input and output interrupt levels.

I/O SYSTEM-SUPPLIED ROUTINES

The I/O System supplies each of the highest-level random access device driver procedures. These procedures are:

INIT$IO	Initialize I/O procedure
FINISH$IO	Finish I/O procedure
QUEUE$IO	Queue I/O procedure
CANCEL$IO	Cancel I/O procedure

Use these names when creating DUIBs for devices that make use of the random access driver.

In addition to these procedures, the I/O System contains a NOTIFY procedure that drivers should call when a unit goes off-line. In particular, NOTIFY should be called by a device driver when a diskette is released from a drive that is attached. The device driver should declare the NOTIFY procedure as follows:

```
NOTIFY:  PROCEDURE (UNIT, DDATA$P) EXTERNAL;
    DECLARE
        UNIT BYTE,
        DDATA$P   POINTER;
END NOTIFY:
```

where:

UNIT	BYTE containing the unit number of the unit that has gone off-line.
DDATA$P	POINTER to the user portion of the device's data object. The driver will have received this value previously from the I/O System; see the description of the device$start procedure later in this chapter.

After the call to NOTIFY, whenever ther is an I/O request for the device (other than detach$device), the device driver should update the I/O result segment by placing an E$IO condition code in its STATUS field and IO$OPRINT in its UNIT$STATUS field. This must remain the case until the device is detached and reattached.

Each of the random access driver procedures calls additional procedures, some of which the I/O System supplies, and some of which you must supply. The following sections describe the routines that you must supply.

USER-SUPPLIED ROUTINES

The routines provided by the I/O System for the random access device driver make up the majority of the routines needed for this device driver. However, these routines make calls to device-dependent routines that you must supply. These device-dependent routines include:

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

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

A device start procedure. This procedure must start the device processing any possible I/O function. RAD$QUEUE$IO and RAD$INTERRUPT$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 unacceptable amount of time. RAD$CANCEL$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. RAD$INTERRUPT$TASK calls this procedure.

You can write these routines in either PL/M-86 or assembly language. However, you must adhere to the following guidelines:

- If you use PL/M-86, you must define your routines as reentrant 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 case. 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.

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 in a device information table. Each DUIB contains a pointer to a device information table (refer to Chapter 2). This table contains not only the addresses of the user-supplied routines, but also other device-specific information.

Each DUIB also contains a pointer to a unit information table. You must place specific information about the unit of the device, such as the track size, in the 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 4-1 illustrates this.

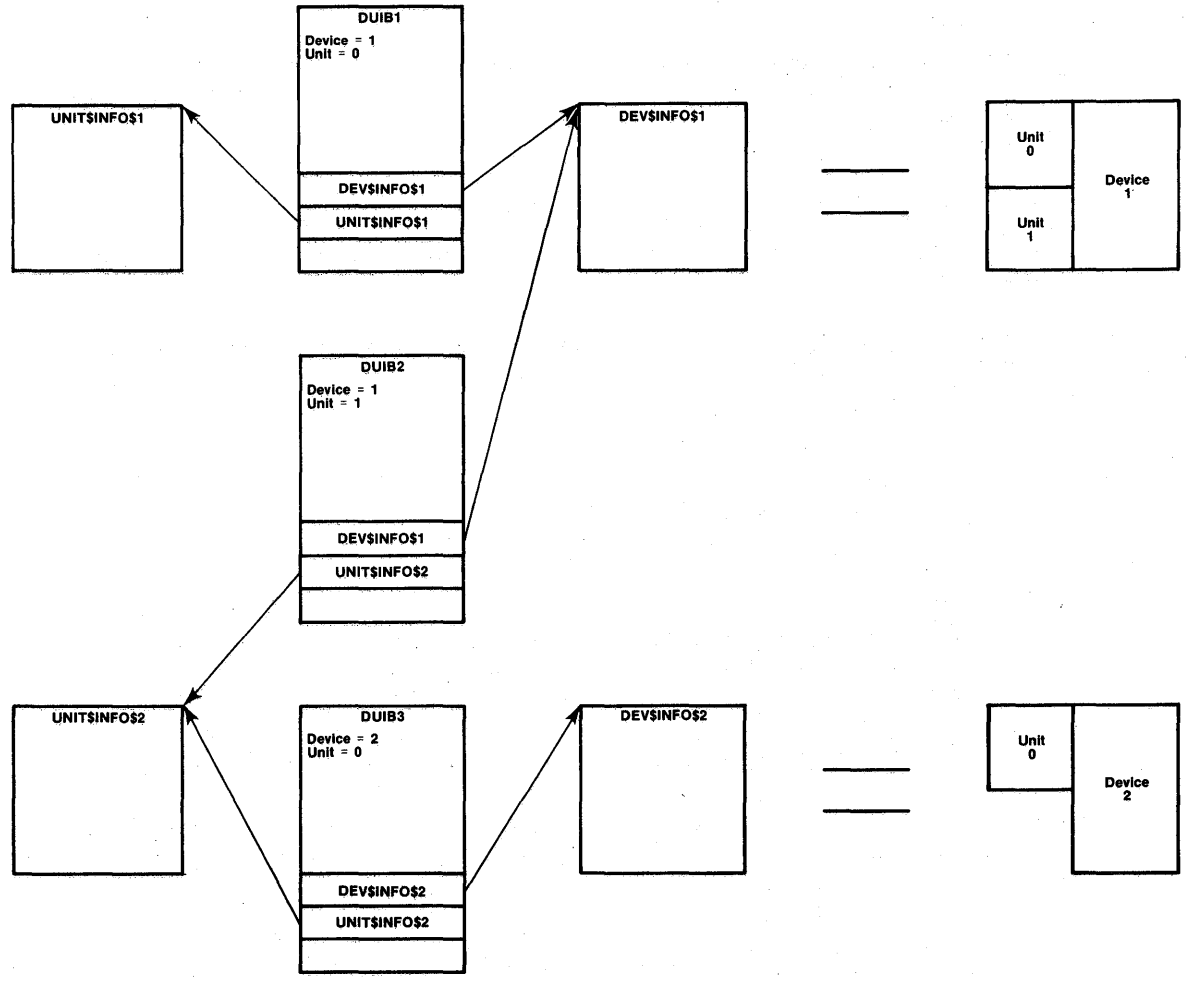


Figure 4-1. DUIBs, Device and Unit Information Tables

Figure 4-1 shows the DUIBs and tables for two devices. The first device, device 1, has two units and thus two DUIBs. Each of the DUIBs for device 1 point to DEV$INFO$1, the device information table for that device. The units of device 1 have slightly different characteristics, however, and the DUIBs point to different unit information tables, each for the appropriate unit. Device 2 has only one unit, and therefore only one DUIB. That DUIB points to DEV$INFO$2, the device information table for device 2. Since this single unit of device 2 has the same characteristics as a unit of device 1, the DUIB can point to the same unit information table as a DUIB for device 1.

The following sections describe the structure of the device and unit information tables for random access devices and the required formats of the user-supplied routines.

DEVICE AND UNIT INFORMATION TABLES FOR RANDOM ACCESS DEVICES

You must place the device-specific information for each random access device in a device information table and the unit-specific information in unit information tables.

The following sections describe the structures of the device and unit information tables for random access devices.

Device Information Table for Random Access Devices

You must create a device information table for each random access device in your system as follows:

```
DECLARE
     RAD$DEVICE$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.

PRIORITY	BYTE specifying the initial priority of the interrupt task. The actual priority of the interrupt task may change due to the fact that the 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 object. This figure should not include the amount needed by the I/O System-supplied procedures, but 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 is described later in this chapter.
DEVICE$FINISH	WORD specifying the start address of a user-written device finish procedure. The format of this procedure is described later in this chapter.
DEVICE$START	WORD specifying the start address of a user-written device start procedure. The format of this procedure is described later in this chapter.
DEVICE$STOP	WORD specifying the start address of a user-written device stop procedure. The format of this procedure is described later in this chapter.
DEVICE$INTERRUPT	WORD specifying the start address of a user-written device interrupt procedure. The format of this procedure is described later in this chapter.

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

Unit Information Table for Random Access Devices

You must create a unit information table for each different type of unit in your system. Although each DUIB must point to a unit information table, several 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$SIZE — WORD specifying the size in bytes of a single track of a volume on the unit. If the device controller supports reading and writing across track boundaries, place a zero in this field. If you specify a zero for this field, the I/O System-supplied procedures place a 32-bit sector number in the HIGHDEVLOC and LOWDEVLOC fields of the IORS. If you specify a nonzero value for this field, the I/O System-supplied procedures guarantee that read and write requests do not cross track boundaries. They do this by placing the sector number in the low order word of the DEV$LOC field of the IORS and the track number in the high order word before calling a user-written device start procedure. Instructions for writing a device start procedure are contained later in this chapter.

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

RESERVED — Reserved WORD.

Depending on the requirements of your device, you can append additional information to this RAD$UNIT$INFO structure. For example, the iSBC 204 driver requires that drive characteristics information be appended to this structure.

You must create unit information tables as a part of the I/O System configuration process. The iRMX 86 CONFIGURATION GUIDE describes this procedure.

DEVICE INITIALIZATION PROCEDURE

The RAD$INIT$IO procedure calls the user-written device initialization procedure in order to initialize the device. The format of the call to the 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$p	POINTER to the user portion of the device's data object. You must specify the size of this portion in the device information table for this device. The device initialization procedure can use this data area for whatever purposes it chooses. Possible uses for this data area include local flags and buffer areas.
status$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 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 RAD$FINISH$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$finish	Name of the device finish procedure. You can use any name for this procedure, as long as long as it doesn't conflict with other procedure names and you include the name in the device information table.
duib$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 object. The device finish procedure should obtain, from the data object, tokens for any objects that other user-written procedures may have created, and delete these objects.

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. DEFAULT$FINISH is normally used when the default initialization procedure DEFAULT$INIT is used.

DEVICE START PROCEDURE

Both RAD$QUEUE$IO and the interrupt task make calls to the device start procedure in order to start an I/O function. RAD$QUEUE$IO calls this procedure on the first I/O request for a device and on other I/O requests when the request queue is empty. The interrupt task calls the device start procedure after it finishes one I/O request if there are more I/O requests on the queue or if the driver has broken up the request into multiple requests. The format of the call to the device start procedure is as follows:

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

where:

device$start	Name of the device start procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include this name in the device information table.
iors$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. If you specified a non-zero track size for the device in the unit information table, the DEV$LOC field of the IORS contains the device location in the following form: LOWDEVLOC sector number HIGHDEVLOC track number If you specified a zero for the track size in the unit information table, the DEV$LOC field contains a 32-bit sector number.
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 object. 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 add 128 to the value 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$STATUS field to indicate the appropriate unit status. The lower four bits of the IORS.UNIT$STATUS 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 drivers return the device's result byte in these remaining bits). If the device start procedure sets the IORS.UNIT$STATUS field to indicate an IO$SOFT condition, the random access driver will retry the I/O operation until the retry limit specified in the unit information table is reached. (The retry limit is described in the "Unit Information Table for Random Access Devices" section of this chapter.) The device start procedure must also set the IORS.DONE field to TRUE, indicating that the request is finished because of the error. If no error occurs, 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 is finished, either because of an error or because the request has been successfully completed, the device start procedure 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 start 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 random access driver support routines will wait until the device sends another interrupt before determining that the request is actually finished.

DEVICE STOP PROCEDURE

The RAD$CANCEL$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$stop	Name of the device stop procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the device information table.
iors$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 the device's data object. The device stop procedure can use this data area to set flags, 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

The 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$interrupt	Name of the device interrupt procedure. You can use any name for this procedure, as long as it does not conflict with other procedure names and you include the name in the device information table.
iors$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 object. 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 object, 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 IORS.UNIT$STATUS field should be set as indicated in the "IORS Structure" section of Chapter 2. The remaining bits of the field can be set to any value (for example, the iSBC 204 and 206 drivers return the device's result byte in these remaining bits). If the device interrupt procedure sets the IORS.UNIT$STATUS field to indicate an IO$SOFT condition, the random access driver will retry the I/O operation until the retry limit specified in the unit information table is reached. (The retry limit is described in the "Unit Information Table for Random Access Devices" section of this chapter.) The device interrupt procedure 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 5. 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 objects, implementing a request queue, and creating an interrupt handler. You can do this in any manner that you choose as long as you divide your driver up into the following four procedures that the I/O System can call:

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

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

A Queue I/O Procedure. 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.

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

The I/O System provides these four procedures for random access and common device drivers. However, if your device does not fit into either of those categories, you must create the procedures yourself.

You can write these routines in either PL/M-86 or assembly language. However, you must adhere to the following guidelines:

- If you use PL/M-86, you must define your routines as reentrant 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.

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 procedure for creating DUIBs is contained in the iRMX 86 CONFIGURATION GUIDE.

The following sections describe the format of the I/O System calls to these four procedures. You must make your procedures conform to these formats.

INITIALIZE I/O PROCEDURE

The 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. In this 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, the Initialize I/O procedure should create it. 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$io procedure can place a token for a data object, if the device driver needs such a data object. If the device driver requires that a data object be associated with a device (to contain the I/O queue, DUIB addresses, or status information), the init$io procedure should create this object and save a token for it via this pointer. If the driver does not need such a data object, 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$OK condition code, it must delete any objects that it has created and leave all data fields with exactly the same information that they contained prior to the call to init$io.

FINISH I/O PROCEDURE

The I/O System calls the Finish I/O procedure after the user makes an RQAPHYSICAL$DETACH$DEVICE system call to detach the last unit of a device. When the I/O System determines that all units of a device have been detached, it calls the Finish I/O procedure to perform any necessary final processing on the device. The Finish I/O procedure must delete all objects 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 the last device-unit 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 a token for the data object originally created by the init$io procedure. The finish$io procedure must delete this object 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 a token for an IORS. This IORS describes the request. When the request is finished, the driver (though not necessarily the queue$io procedure) must fill out the status fields and send the IORS to the response mailbox indicated in the IORS. Chapter 2 describes the format of the IORS. It lists the information that the I/O System supplies when it passes the IORS to the queue$io procedure and indicates the fields of the IORS that the device driver must fill out.
duib$p	POINTER to the DUIB of the device-unit for which the request is intended.
ddata$t	WORD containing a token for the data object originally created by the init$io procedure. The queue$io procedure can place any necessary information in this object in order to update the request queue or status fields.

CANCEL I/O PROCEDURE

The I/O System calls the Cancel I/O procedure in order to cancel one or more previously queued I/O requests. This is done under one 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 a finite amount of time, the Cancel I/O procedure does not have to stop the device itself, only remove requests from the queue.

Currently, only the physical file driver calls the Cancel I/O procedure.

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 DUIB of device-unit for which the request cancellation is intended.
ddata$t	WORD containing a token for the data object originally created by the init$io procedure. This object may contain the request queue.

CHAPTER 6. LINKING DRIVER ROUTINES TO THE I/O SYSTEM

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 may 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, this chapter contains a brief description of the 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.
IOS.LIB	I/O System library.
IOOPT1.LIB	I/O System options 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 I/O System calls INIT$IO when an application task makes an RQAPHYSICAL$ATTACH$DEVICE system call and there are no units of the device currently attached. INIT$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
- A pointer to the location where INIT$IO must return a token for a data segment that it creates
- A pointer to the location where INIT$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.

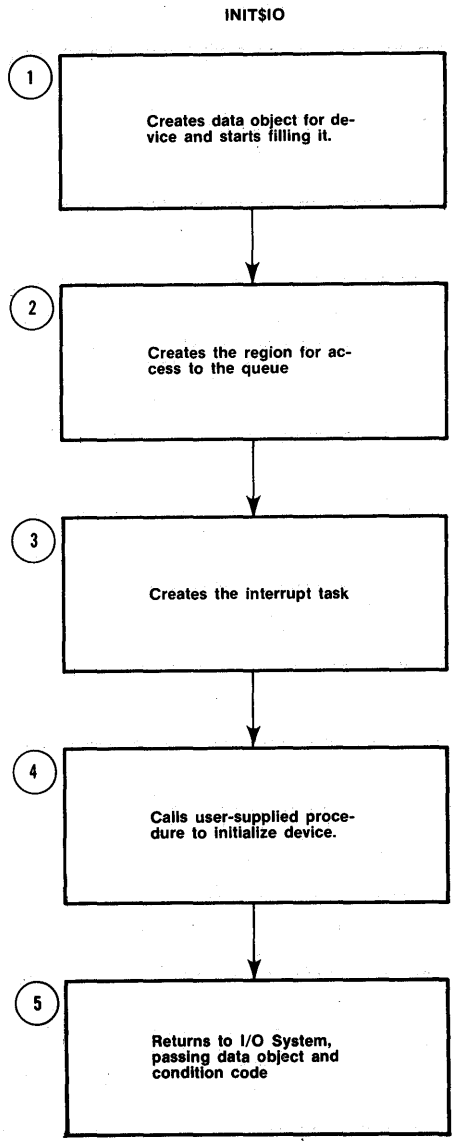


Figure A-1. Common Device Driver Initialize I/O Procedure

1. It creates a data segment which will be used by all of the procedures in the device driver. The size of this segment 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 segment and eventually places the following information in this segment:

- The value of the DS (data segment) register
- A token for a region
- The address of the DUIB for this device-unit
- A token 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 object 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 object.

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 object and a condition code which indicates the success of the initialize operation.

FINISH$IO PROCEDURE

The 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. It 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 segment 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 3 for information on how to write this 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 segment 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.

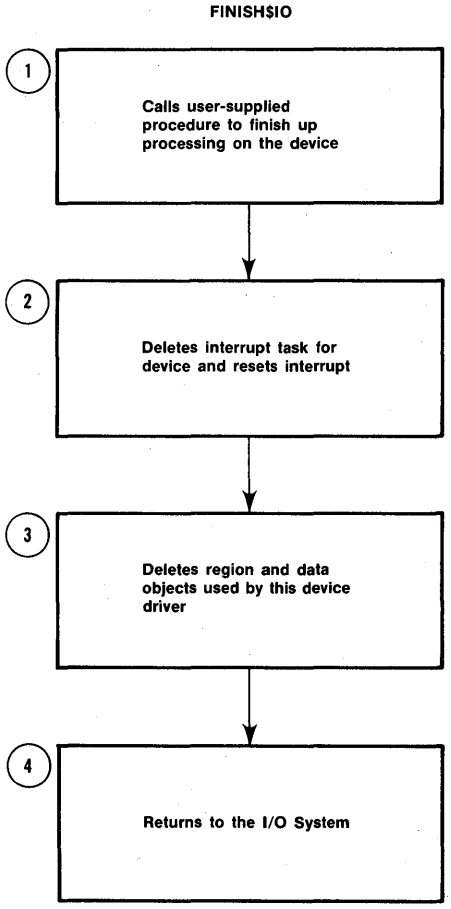


Figure A-2. Common Device Driver Finish I/O Procedure

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 for the IORS
- A pointer to the DUIB
- A token 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 various fields in the IORS to indicate 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 these fields.

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 3. 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.

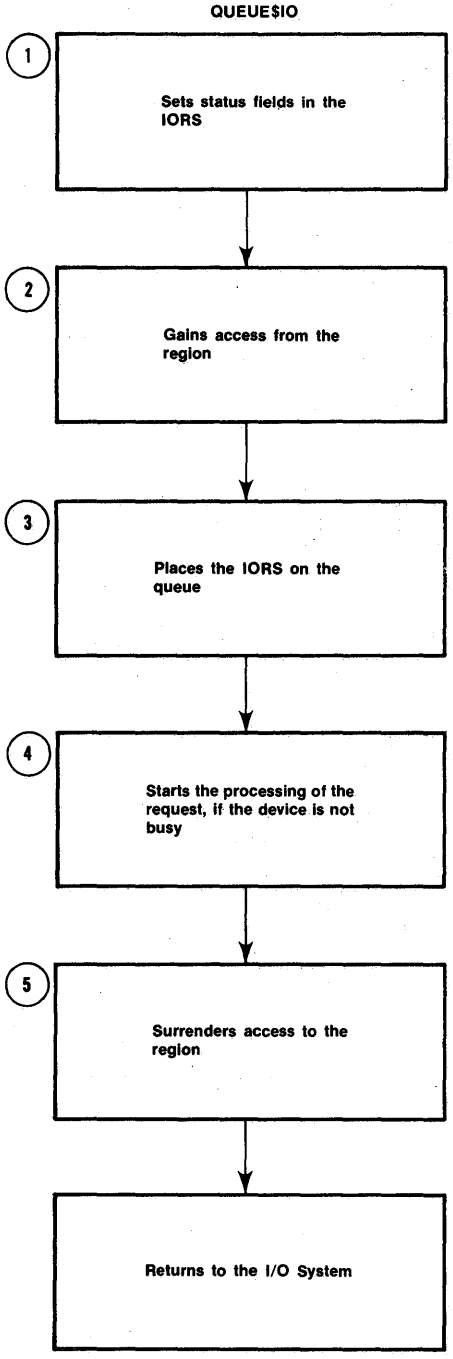


Figure A-3. Common Device Driver Queue I/O Procedure

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 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 for the device data object

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 IORs, 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 3 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 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 dealth with.

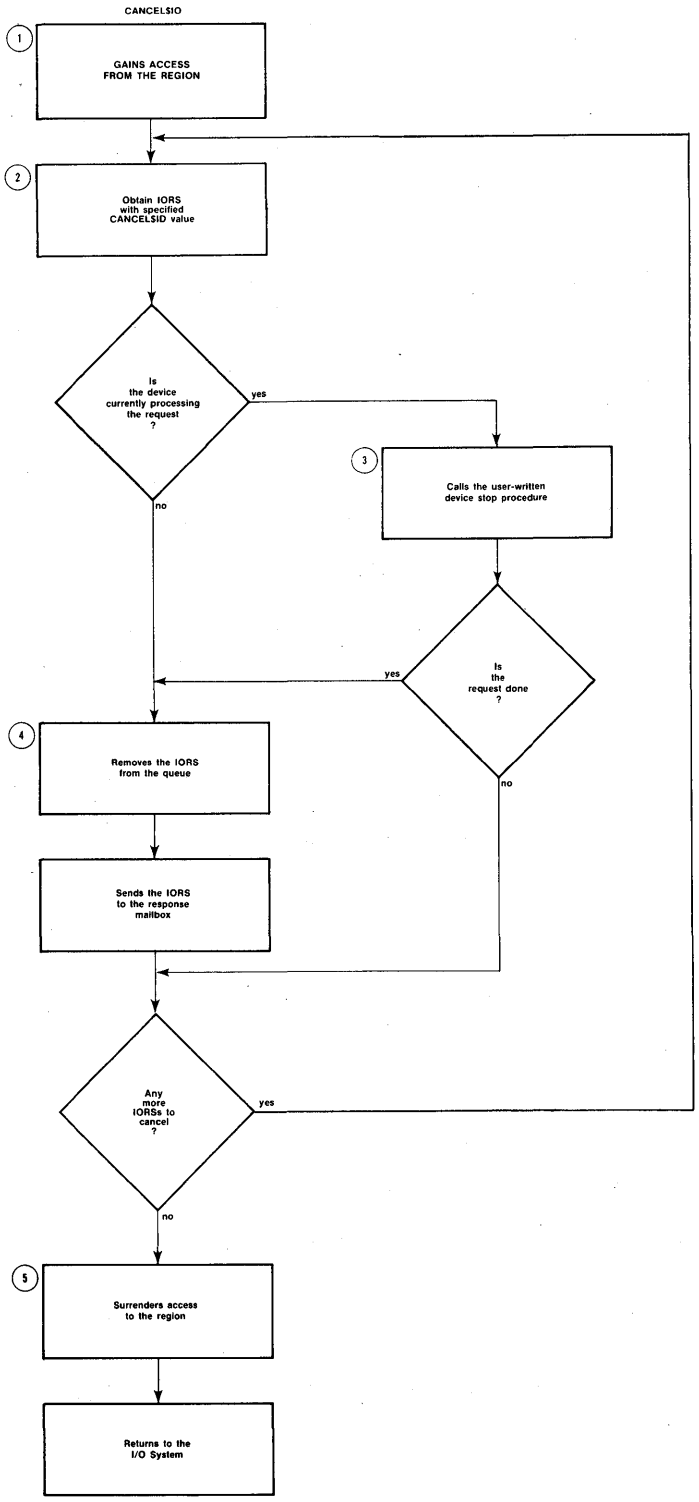


Figure A-4. Common Device Driver Cancel I/O Procedure

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 for the device data object. 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 object, 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 object) and sets the DS register to the value listed in the field of the data object. Thus the task has the correct value in its DS register and it has the address of the data object. This is the mechanism that is used to pass the device's data object 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.

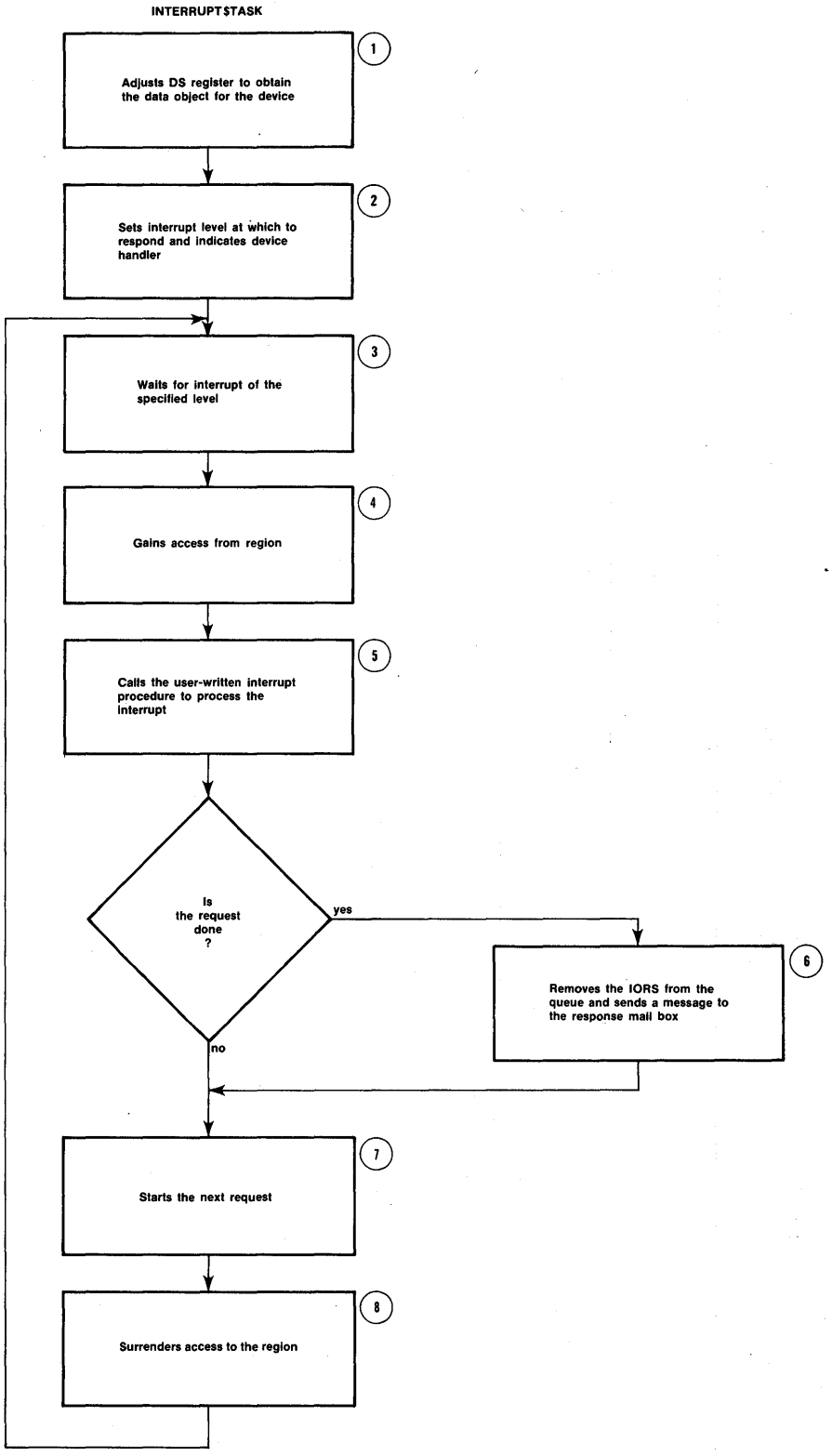


Figure A-5. Common Device Driver Interrupt Task

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 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, it then surrenders access to the queue, allowing other routines to modify the queue, and loops back to wait for another interrupt.

APPENDIX B. FOLD-OUT FIGURES

This appendix contains fold-out figures which have been referred to elsewhere in this manual. To use the figures in this appendix, unfold them and refer to them while you read related text from the previous chapters.

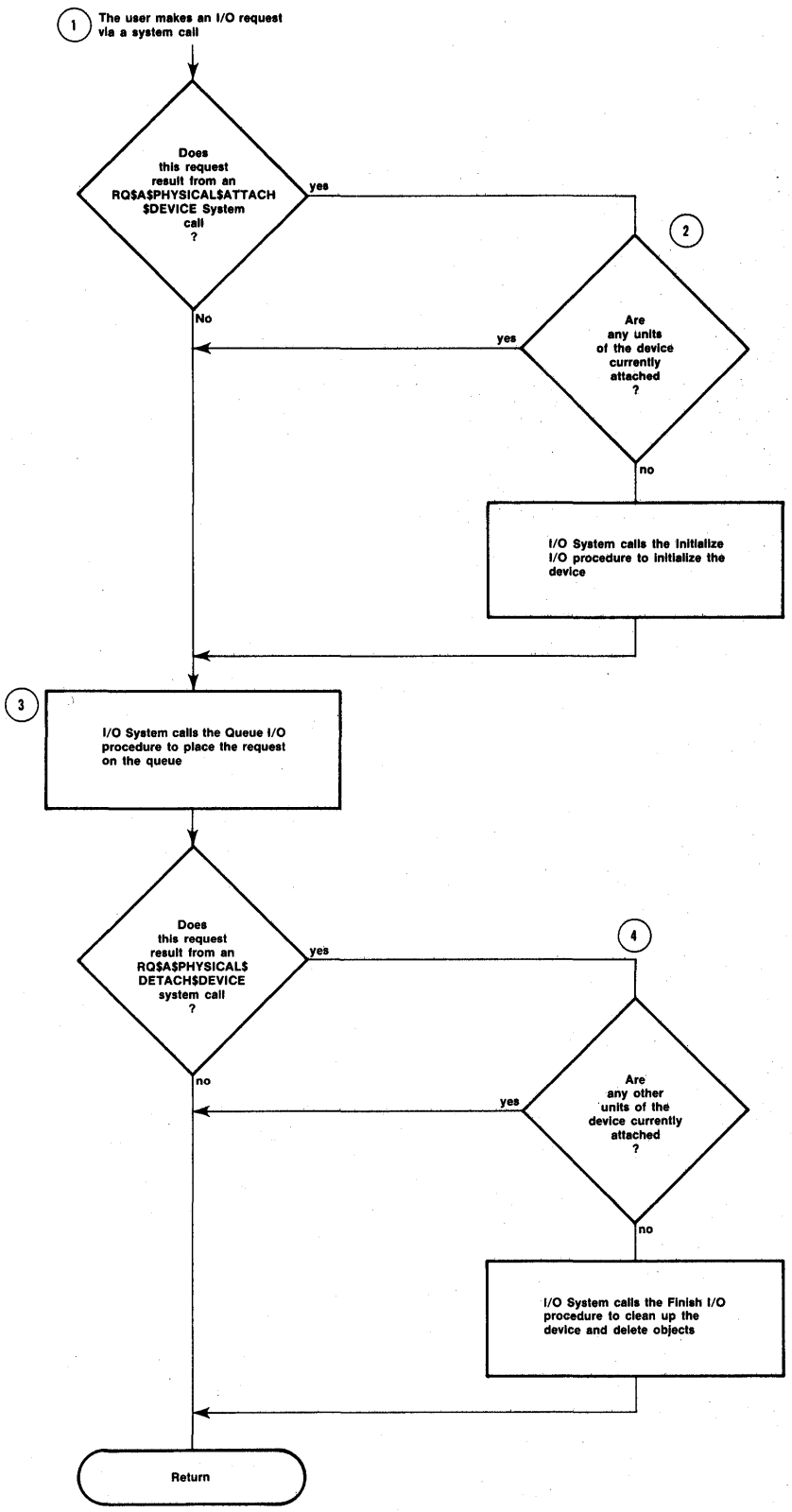


Figure B-1. Calling the Device Driver Procedures

INDEX

For most topics with multiple-page references, the primary reference is underscored.

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 _____

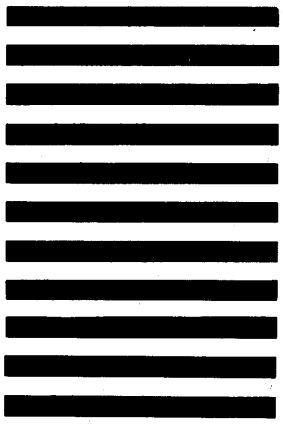
CITY _____ STATE _____ ZIP CODE _____

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.