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

CONTENTS

PAGE

CHAPTER 1
INTRODUCTION
I/O Devices and Device Drivers.. 1-2
I/O Requests.. 1-3
Types of Device Drivers.. 1-3
How to Read This Manual.. 1-4

CHAPTER 2
DEVICE DRIVER INTERFACES
I/O System Interfaces.. 2-1
Device-Unit Information Block (DUIB)................................. 2-2
DUIB Structure... 2-2
Using the DUIBs.. 2-7
Creating DUIBs... 2-8
I/O Request/Result Segment (IORS).................................... 2-9
Device Interfaces.. 2-16

CHAPTER 3
CATEGORIES AND PROPERTIES OF DEVICES AND DRIVERS
Categories of Devices.. 3-1
Common Devices.. 3-1
Random Access Devices.. 3-1
Terminal Devices... 3-2
Custom Devices... 3-2
I/O System-Supplied Routines for Common and Random Access Device Drivers... 3-2
I/O System Algorithm For Calling the Device Driver Procedures...... 3-4
Required Data Structures... 3-7
Device Information Table... 3-8
Unit Information Table... 3-10
Relationships Between I/O Procedures and I/O Data Structures..... 3-12
Device Data Storage Area... 3-13
Writing Drivers For Use With Both iRMX™ 86- and iRMX™ 88-Based Systems... 3-14

CHAPTER 4
I/O REQUESTS
I/O System Responses to I/O Requests................................. 4-1
Attach Device Requests... 4-1
Detach Device Requests... 4-2
Read, Write, Open, Close, Seek, and Special Requests............. 4-2
Cancel Requests.. 4-2
DUIB and IORS Fields Used By Device Drivers......................... 4-3

CONTENTS (continued)

PAGE

CHAPTER 5
WRITING COMMON OR RANDOM ACCESS DEVICE DRIVERS
Introduction to Procedures that Device Drivers Must Supply......... 5-1
Device Initialization Procedure.................................... 5-2
Device Finish Procedure.. 5-2
Device Start Procedure... 5-3
Device Stop Procedure.. 5-4
Device Interrupt Procedure... 5-5
Procedures that iRMX™ 86 Random Access Drivers Must Call........... 5-6
NOTIFY Procedure... 5-6
SEEK$COMPLETE Procedure.. 5-7
Procedures for Other Long-Term Operations.......................... 5-8
BEGIN$LONG$TERM$OP Procedure....................................... 5-8
END$LONG$TERM$OP Procedure... 5-9
GET$IORS Procedure... 5-10
Formatting Considerations.. 5-11

CHAPTER 6
WRITING A CUSTOM DEVICE DRIVER
Initialize I/O Procedure... 6-1
Finish I/O Procedure... 6-2
Queue I/O Procedure.. 6-3
Cancel I/O Procedure... 6-4
Implementing a Request Queue....................................... 6-5

CHAPTER 7
TERMINAL DRIVERS
Terminal Support Code.. 7-1
Data Structures Supporting Terminal I/O............................ 7-2
DUIB... 7-2
Device Information Table... 7-3
Unit Information Table... 7-6
Terminal Controller Data and Terminal Unit Data.................... 7-4
Terminal Support Code (TSC) Data Area.............................. 7-11
Procedures that Terminal Drivers Must Supply....................... 7-17
Terminal Initialization Procedure.................................. 7-18
Terminal Finish Procedure.. 7-18
Terminal Setup Procedure... 7-19
Terminal Answer Procedure.. 7-20
Terminal Hangup Procedure.. 7-21
Terminal Check Procedure... 7-22
Terminal Output Procedure.. 7-24
Additional Information for Buffered Devices........................ 7-25
Procedures' Use of Data Structures................................. 7-27

CHAPTER 8
BINDING A DEVICE DRIVER TO THE I/O SYSTEM
Using the iRMX™ 86 Interactive Configuration Utility............... 8-1
Using the iRMX™ 88 Interactive Configuration Utility............... 8-4

CONTENTS (continued)

PAGE

APPENDIX A
RANDOM ACCESS DRIVER SUPPORT ROUTINES
INIT$IO Procedure.. A-1
FINISH$IO Procedure.. A-3
QUEUE$IO Procedure... A-5
CANCEL$IO Procedure.. A-6
Interrupt Task (INTERRUPT$TASK)............................ A-9

APPENDIX B
EXAMPLES OF DEVICE DRIVERS................................. B-1

FIGURES

1-1. Communication Levels.................................. 1-1
1-2. Device Numbering...................................... 1-2
2-1. Attaching Devices..................................... 2-8
3-1. Interrupt Task Interaction............................ 3-4
3-2. How the I/O System Calls the Device Driver Procedures...... 3-6
3-3. DUIBs, Device and Unit Information Tables............. 3-7
3-4. Relationships Between I/O Procedures and I/O Data Structures.. 3-12
6-1. Request Queue... 6-6
7-1. Software Layers Supporting Terminal I/O............... 7-2
7-2. TSC Data Area... 7-12
8-1. Example IDEVCF.A86 File............................... 8-2
8-2. Example User Devices Screen........................... 8-4
A-1. Random Access Device Driver Initialize I/O Procedure....... A-2
A-2. Random Access Device Driver Finish I/O Procedure........... A-4
A-3. Random Access Device Driver Queue I/O Procedure............ A-6
A-4. Random Access Device Driver Cancel I/O Procedure........... A-8
A-5. Random Access Device Driver Interrupt Task................. A-10

TABLES

4-1. DUIB and IORS Fields Used by Common Device Drivers......... 4-4
4-2. DUIB and IORS Fields Used by Random Access Device Drivers.. 4-5
4-3. DUIB and IORS Fields Used by Custom Device Drivers......... 4-6
7-1. Uses of Fields in Terminal Driver Data Structures.......... 7-28

***

CHAPTER 1
INTRODUCTION

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

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

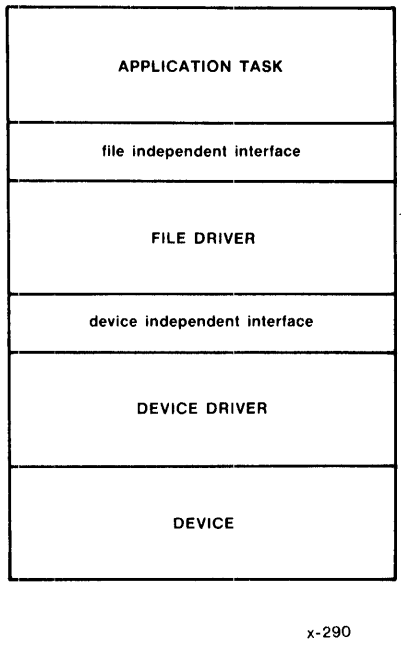

Figure 1-1. Communication Levels

The I/O System provides a standard interface between file drivers and device drivers. To a file driver, a device is merely a standard block of data in a table. 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 change without extensive modifications to the software. Instead of modifying entire file drivers when you want to change devices, you need only substitute a different device driver and modify the table.

- The I/O System can support a greater range of devices. It can support any device, as long as you supply a device 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 unique device number. Units are identified by unit number and by device-unit number. The device number identifies the controller among all the controllers in the system, the unit number identifies the unit within the device, and the unique 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.

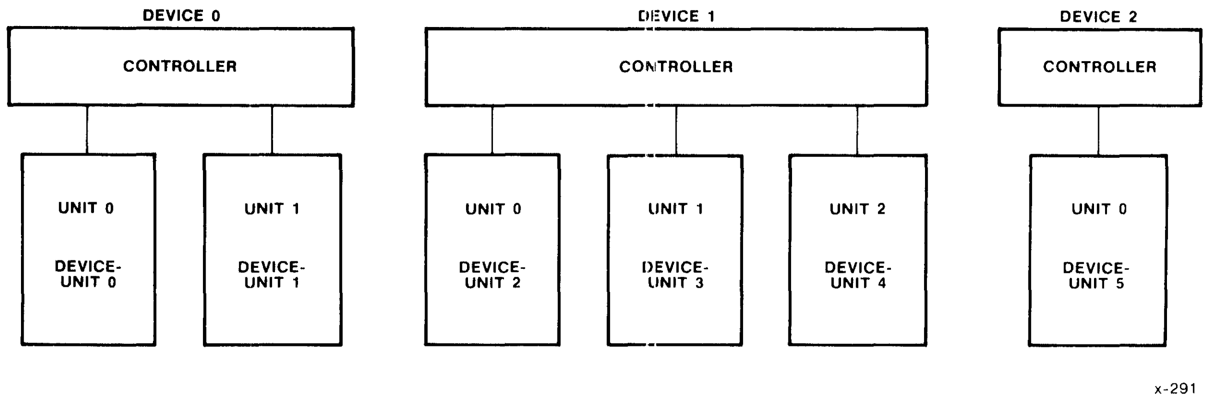

Figure 1-2. Device Numbering

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 215 controllers are two separate devices and each has its own device driver. However, these device drivers can share common code.)

I/O REQUESTS

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

Read
Write
Seek
Special
Attach device
Detach device
Open
Close

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

TYPES OF DEVICE DRIVERS

The I/O System supports four types of device drivers: custom, common, random access, and terminal. A custom device driver is one that the user creates in its entirety. This type of device driver can assume any form and can 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, random access, and terminal device driver types. These support routines provide a queueing mechanism, an interrupt handler, and other features needed by common, random access, and terminal devices. If your device fits into the common, random access, or terminal device classification, you need to write only the specialized, device-dependent procedures and interface them to the ones provided by the I/O System to create a complete device driver.

HOW TO READ THIS MANUAL

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

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

***

CHAPTER 2
DEVICE DRIVER INTERFACES

Because a device driver is a collection of software routines that manages a device at a basic level, it must transform general instructions from the I/O System into device-specific instructions which it then sends to the device itself. Thus, a device driver has two types of interfaces:

- An interface to the I/O System, which is the same for all device drivers.

- An interface to the device itself, which varies according to device.

This chapter discusses these interfaces.

I/O SYSTEM INTERFACES

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

DEVICE-UNIT INFORMATION BLOCK (DUIB)

The DUIB is an interface between a device driver and the I/O System, in the sense that the DUIB contains the addresses of one of the following routines:

- The device driver routines (in the case of custom device drivers).

- The device driver support routines (in the case of terminal drivers, common drivers, and random access drivers).

By accessing the DUIB for a unit, the I/O System can call the appropriate device driver/device driver support routine. All devices, no matter how diverse, use this standard interface to the I/O System. You must provide a DUIB for each device-unit in your hardware system. You supply the information for your DUIBs as part of the configuration process.

DUIB Structure

This section lists the elements that make up a DUIB. When creating DUIBs for iRMX 86 applications, code them in the format shown here (as assembly-language structures). The iRMX 86 Interactive Configuration Utility (ICU) includes your DUIB file in the assembly of IDEVCF.A86 (a Basic I/O System configuration file). IDEVCF.A86 contains the definition of the structure.

Unlike the iRMX 86 ICU, the iRMX 88 ICU prompts you for some fields in the DUIB structure. The ICU automatically fills in the other fields, depending upon factors such as the type of device you are configuring. The iRMX 88 ICU generates the DUIBs and places them in the device configuration source file.

```
DEFINE_DUIB  <
&    NAME (14),                        ; byte (14)
&    FILE$DRIVERS,                     ; word
&    FUNCTS,                           ; byte
&    FLAGS,                            ; byte
&    DEV$GRAN,                         ; word
&    DEV$SIZE,                         ; dword
&    DEVICE,                           ; byte
&    UNIT,                             ; byte
&    DEV$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
&    FIXED$UPDATE,                     ; byte (iRMX 86 DUIB only)
&    MAX$BUFFERS,                      ; byte (iRMX 86 DUIB only)
&    RESERVED,                         ; byte (iRMX 86 DUIB only)
&    >
```

where:

NAME	A 14-BYTE array specifying the name of the DUIB. This name uniquely identifies the device-unit to the I/O System. Use only the first 13 bytes. The fourteenth is used by the I/O System.

You supply the name when configuring your application system. If you are an iRMX 86 user, you specify the DUIB name when attaching a unit via the RQ$A$PHYSICAL$ATTACH$DEVICE system call. Device drivers can ignore this field.

For the iRMX 88 Executive, the DUIB name is the device name portion of the name$p parameter for the DQ$ATTACH or the DQ$CREATE system calls.

FILE$DRIVERS — WORD specifying file driver validity. Setting bit number "i" of this word implies that the corresponding file driver can attach this device-unit. Clearing bit number "i" implies that the file driver cannot attach this device-unit.

The low-order bit is bit 0. The bits are associated with the file drivers as follows:

Bit "i"	File Driver
0	physical
1	stream (iRMX 86 only)
3	named

The remaining bits 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 the device-unit supports the corresponding function. Clearing bit number "i" implies that the device-unit does not support the function. The low-order bit is bit 0. The bits are associated with the functions as follows:

Bit "i"	Function
0	read
1	write
2	seek
3	special
4	attach device
5	detach device
6	open
7	close

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

This field is used for informational purposes only. Setting or clearing 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 any I/O function, either by performing the function or by returning a condition code indicating the inability of the device to perform that function. However, to provide accurate status information, this field should indicate the device's ability to perform the I/O functions. Device drivers can ignore this field.

FLAGS

BYTE specifying characteristics of diskette devices. The significance of the bits is as follows, with bit 0 being the low-order bit:

Bit	Meaning
0	0 = bits 1-7 not significant 1 = bits 1-7 significant
1	0 = single density; 1 = double density
2	0 = single sided; 1 = double sided
3	0 = 8-inch diskettes 1 = 5 1/4-inch diskettes
4	0 = standard diskette, meaning that track 0 is single-density with 128-byte sectors 1 = not a standard diskette or not a diskette
5-7	reserved

If bit 0 is set to 1, then a driver for the device can read track 0 when asked to do so by the I/O System.

DEV$GRAN

WORD specifying the device granularity, in bytes. This parameter applies to random access devices. It specifies the minimum number of bytes of information that the device reads or writes in one operation.If the device is a disk or magnetic bubble device, you should set this field equal to the sector size for the device. Otherwise, set this field equal to zero.

DEV$SIZE

DWORD specifying the number of bytes of information that the device-unit can store.

DEVICE

BYTE specifying the device number of the device with which this device-unit is associated. Device drivers can ignore this field.

UNIT	BYTE specifying the unit number of this device-unit. This distinguishes the unit from the other units of the device.
DEV$UNIT	WORD specifying the device-unit number. This number distinguishes the device-unit from the other units in the entire hardware system. Device drivers can ignore this field.
INIT$IO	WORD specifying the address of the Initialize I/O procedure associated with this unit. When creating the DUIB, use the procedure name as a variable to supply this information. Device drivers can ignore this field.
FINISH$IO	WORD specifying the address of the Finish I/O procedure associated with this unit. When creating the DUIB, use the procedure name as a variable to supply this information. Device drivers can ignore this field.
QUEUE$IO	WORD specifying the address of the Queue I/O procedure associated with this unit. When creating the DUIB, use the procedure name as a variable to supply this information. Device drivers can ignore this field.
CANCEL$IO	WORD specifying the address of the Cancel I/O procedure associated with this unit. When creating the DUIB, use the procedure name as a variable to supply this information. Device drivers can ignore this field.
DEVICE$INFO$P	POINTER to a structure which contains additional information about the device. The common, random access, and terminal device drivers require, for each device, a Device Information Table, in a particular format. This structure is described in Chapter 3. If you are writing a custom driver, you can place information in this structure depending on the needs of your driver. Specify a zero for this parameter if the associated device driver does not use this field.
UNIT$INFO$P	POINTER to a structure that contains additional information about the unit. Random access and terminal device drivers require this Unit Information Table in a particular format. Refer to Chapter 3 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$TIMEOUT	WORD specifying the number of system time units that the I/O System must wait before writing a partial sector after processing a write request for a disk device. In the case of drivers for devices that are neither disk nor magnetic bubble devices, set this field to 0FFFFH during configuration. This field applies only to the device for which this is a DUIB, and is independent of updating that is done either because of the value in the FIXED$UPDATE field of the DUIB or by means of the A$UPDATE system call of the I/O System. Device drivers can ignore this field.
NUM$BUFFERS	WORD which, if not zero, both specifies that the device is a random access device and indicates the number of buffers the I/O System allocates. The I/O System uses these buffers to perform data blocking and deblocking operations. That is, it guarantees that data is read or written beginning on sector boundaries. If you desire, the random access support routines can also 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 3). 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.
FIXED$UPDATE	BYTE indicating whether the fixed update option was selected for the device when the application system was configured. This option, when selected, causes the I/O System to finish any write requests that had not been finished earlier because less than a full sector remained to be written. Fixed updates are performed throughout the entire system whenever a time interval (specified during configuration) elapses. This is independent of the updating that is indicated for a particular device (by the UPDATE$TIMEOUT field of the DUIB) or the updating of a particular device that is indicated by the A$UPDATE system call of the I/O System. A value of 0FFH indicates that fixed updating has been selected for this device, and a value of zero indicates that it has not been selected. Device drivers can ignore this field. The FIXED$UPDATE field is not present in the iRMX 88 DUIB.

MAX$BUFFERS	BYTE specifying the maximum number of buffers that the Extended I/O System (of the iRMX 86 Operating System) can allocate for a connection to this device when the connection is opened by a call to S$OPEN. The value in this field is specified during configuration. Device drivers can ignore this field. The MAX$BUFFERS field is not present in the iRMX 88 DUIB.
RESERVED	BYTE reserved for future use. The RESERVED field is not present in the iRMX 88 DUIB.

Using the DUIBs

To use the I/O System to communicate with files on a device-unit, you must first attach the unit. If you are an iRMX 88 user, attaching the unit occurs automatically when you first attach or create a file on the unit. If you are an iRMX 86 user, you attach the unit by invoking the RQ$A$PHYSICAL$ATTACH$DEVICE system call (refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for a description of this system call).

When you attach a unit, the I/O System assumes that the device-unit identified by the device name field of the DUIB has the characteristics identified in the remainder of the DUIB. Thus, whenever the application software makes an I/O request via the connection to the attached device-unit, the I/O System ascertains the characteristics of that unit by examining the associated DUIB. The I/O System looks at the DUIB and calls the appropriate device driver/device driver support routines listed there to process the I/O request.

If you want the I/O System to assume different characteristics at different times for a particular device-unit, you can supply multiple DUIBs, each containing identical device number, unit number, and device-unit number parameters, but different DUIB name parameters. Then you can select one of these DUIBs by specifying the appropriate dev$name parameter in the RQ$A$PHYSICAL$ATTACH$DEVICE system call (for iRMX 86 users) or the appropriate device name when calling DQ$ATTACH or DQ$CREATE (for iRMX 88 users.) However, before you can switch the DUIBs for a unit, you must detach the unit.

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

NOTE

For iRMX 86 systems only, when the I/O System accesses a device containing named files, it obtains information such as granularity, density, size (5-1/4" or 8" for diskettes), or the number of sides (single or double) from the volume label. Therefore it is not necessary to supply a different DUIB for every kind of volume you intend to use. However, for iRMX 86 applications, you must supply a separate DUIB for every kind of volume you intend to format via the FORMAT Human Interface command.

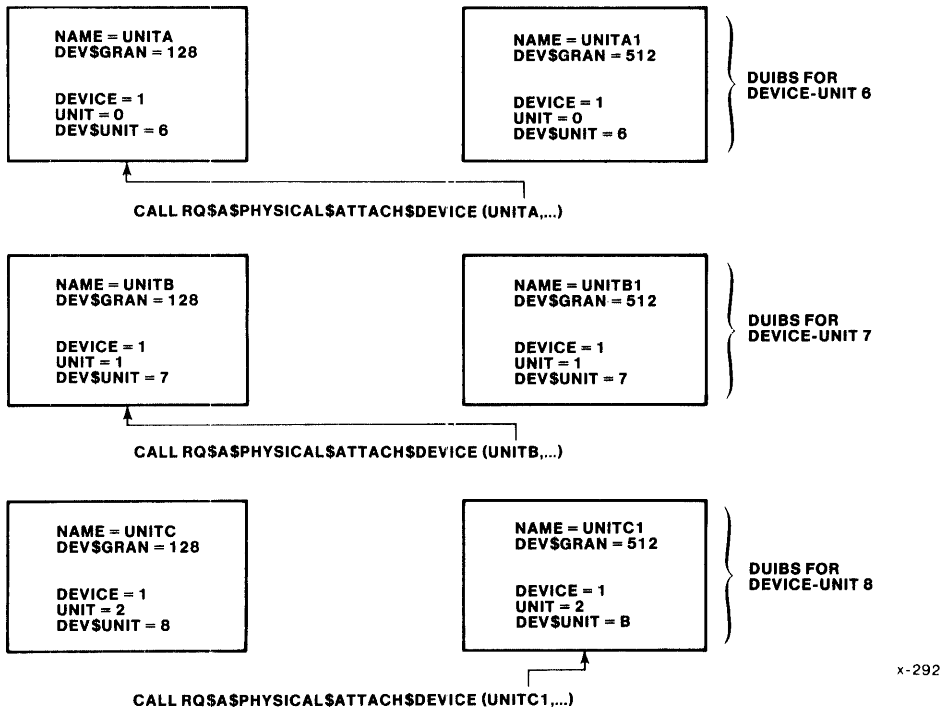

Figure 2-1. Attaching Devices

Creating DUIBs

During interactive configuration, you must provide the information for all of the DUIBs. The configuration file, which the ICU produces, sets up the DUIBs when it executes. Observe the following guidelines when supplying DUIB information:

- Specify a unique name for every DUIB, even those that describe the same device-unit.
- For every device-unit in the hardware configuration, provide information for at least one DUIB. Because the DUIB contains the addresses of the device driver/device driver support 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/device driver support procedures in all of the DUIBs associated with a particular device. There is only one set of device driver/device driver support routines for a given device, and each DUIB for that device must specify this unique set of routines.
- If you write a common or random access device driver, you must supply a Device Information Table for each device. If you write a random access device driver, you must also supply a Unit

  Information Table for each unit. See Chapter 4 for specifications of these tables. If you are using custom device drivers and they require these or similar tables, you must supply them, as well.
- For iRMX 86 systems only, if you write a terminal driver, you must supply terminal device information table for each terminal device driver, as well as a unit information table for each terminal. See Chapter 7 for specifications of these tables.

I/O REQUEST/RESULT SEGMENT (IORS)

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

The IORS is the only mechanism that the I/O System uses to transmit requests to device drivers. The IORS 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,
          DEV$LOC         DWORD,
          BUFF$P          POINTER,
          COUNT           WORD,
          COUNT$FILL      WORD,
          AUX$P           POINTER,
          LINK$FOR        POINTER,
          LINK$BACK       POINTER,
          RESP$MBOX       SELECTOR,
          DONE            BYTE,
          FILL            BYTE,
          CANCEL$ID       SELECTOR,
          CONN$T          SELECTOR); (iRMX 86 IORS only)
```

where:

STATUS — WORD in which the device driver must place the condition code for the I/O operation. The E$OK condition code indicates successful completion of the operation. For a complete list of possible condition codes, see either the iRMX 86 NUCLEUS REFERENCE MANUAL, the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL, and the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL, or the iRMX 88 REFERENCE MANUAL.

UNIT$STATUS — WORD in which the device driver must place additional status information if the status parameter was set to indicate the E$IO condition. The unit status codes and their descriptions are as follows:

Code	Mnemonic	Description
0	IO$UNCLASS	Unclassified error
1	IO$SOFT	Soft error; a retry is possible
2	IO$HARD	Hard error; a retry is impossible
3	IO$OPRINT	Operator intervention is required
4	IO$WRPROT	Write-protected volume
5*	IO$NO$DATA	No data on the next tape record
6*	IO$MODE	A read (or write) was attempted before the previous write (or read) completed

*For iRMX 86 systems only.

The I/O System reserves values 0 through 3 (the least significant four bits) of this field for unit status codes. The high 12 bits of this field can be used for any other purpose that you wish. For example, the iSBC 204 driver places the controller's result byte in the high eight bits of this field. For more information about the data returned by your device controller, refer to the hardware reference manual for your controller.

ACTUAL — WORD which the device driver must update upon completion of an I/O operation to indicate the number of bytes of data actually transferred.

ACTUAL$FILL — Reserved WORD.

DEVICE — WORD into which the I/O System places the number of the device for which this request is intended.

UNIT — BYTE into which the I/O System places the number of the unit for which this request is intended.

FUNCT — BYTE into which the I/O System places the function code for the operation to be performed. Possible function codes are:

Code	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

SUBFUNCT — WORD into which the I/O System places the actual function code of the operation, when the F$SPECIAL function code was placed into the FUNCT field. The value in this field depends upon the file driver to be used with this device. The possible subfunctions and the driver types to which they apply are as follows:

File Driver For Connection	Subfunct Value	Function
Physical*	0	Format track
Stream	0	Query
Stream	1	Satisfy
Physical or Named	2	Notify
Physical	3	Get disk/tape data

File Driver For Connection	Subfunct Value	Function
Physical	4	Get terminal data
Physical	5	Set terminal data
Physical	6	Set signal
Physical	7	Rewind tape
Physical	8	Read tape file mark
Physical	9	Write tape file mark
Physical	10	Retension tape
	11-32767	Reserved for other Intel products

*These functions apply both to iRMX 86 and iRMX 88 systems. The other functions are iRMX 86-specific.

The values from 32768 to 65535 are available for user-written/custom device drivers.

DEV$LOC

DWORD into which the I/O System initially places the absolute byte location on the I/O device where the operation is to be performed. For example, for a write operation, this is the address on the device where writing begins. The I/O System fills out this information when it passes the IORS to the driver support routines.

If the device driver is a random access driver, the random access support routines modify the information in the DEV$LOC field before passing the IORS on to user-written driver procedures listed in Chapter 5. The value that the random access support routines fill out depends upon the TRACK$SIZE field in the unit's Unit Information Table (see Chapter 3).

- If the TRACK$SIZE field is zero, the random access support routines divide the value in DEV$LOC by the device granularity and place that value (the absolute sector number) in the DEV$LOC field.

- If the TRACK$SIZE field is nonzero, the random access support routines use the absolute byte number in DEV$LOC to calculate the track and sector numbers. The routines then place the track number in the high-order WORD (of DEV$LOC) and the sector number in the low-order WORD (of DEV$LOC).

BUFF$P

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

COUNT	WORD which the I/O System sets to indicate the number of bytes to transfer.
COUNT$FILL	Reserved WORD.
AUX$P	POINTER which the I/O System can set to indicate the location of auxiliary data. Normally, the I/O System uses AUX$P to pass or receive the additional data that the various subfunctions of the SPECIAL call require.

The following paragraphs define the particular data structures pointed to by AUX$P. The data structure actually pointed to depends upon the SUBFUNCT field of the IORS.

In a request to format a track on a disk or diskette, FUNCT equals special, SUBFUNCT equals format track, and AUX$P points to a structure of the form:

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

These fields are defined as follows:

track$number	The number of the track to be formatted. Acceptable values are 0 to (number of tracks on the volume - 1).
interleave	The interleaving factor for the track. (That is, the number of physical sectors to advance when locating the next logical sector.) The supplied value, before being used, is evaluated MOD the number of sectors per track.
track$offset	The number of physical sectors to advance when locating the first logical sector on the next track.
fill$char	The byte value with which each sector is to be filled.

NOTE

The rest of the information about the AUX$P field is iRMX 86-specific.

In a request to set up an iRMX 86 mailbox, where the iRMX 86 I/O System is to send an object whenever a door to a flexible disk drive is opened or the STOP button on a hard disk drive is pressed, FUNCT equals special, SUBFUNCT equals notify, and AUX$P points to a structure of the form:

```
DECLARE SETUP$NOTIFY STRUCTURE(
          MAILBOX        SELECTOR,
          OBJECT         SELECTOR);
```

where the fields are defined in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL. Random access drivers do not have to process such requests because they are handled by the I/O System.

In a request to obtain information about iSBC 215 or iSBC 220 (supported) disk devices, FUNCT equals special, SUBFUNCT equals get device characteristics, and AUX$P points to a structure of the form:

```
DECLARE DISK$DRIVE$DATA STRUCTURE(
     CYLINDERS           WORD,
     FIXED               BYTE,
     REMOVABLE           BYTE,
     SECTORS             BYTE,
     SECTOR$SIZE         WORD,
     ALTERNATES          BYTE);
```

where the fields are defined in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

In a request to obtain information about iSBX 217 tape drives (associated with an iSBC 215G board), FUNCT equals special, SUBFUNCT equals get device characteristics, and AUX$P points to a structure of the form:

```
DECLARE TAPE$DRIVE$DATA STRUCTURE(
     TAPE                WORD,
     RESERVED(7)         BYTE);
```

where the fields are defined in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

In a request to read or write terminal mode information for a terminal being driven by a terminal driver, FUNCT equals special, SUBFUNCT equals get terminal attributes (for reading) or set terminal attributes (for writing), and AUX$P points to a structure of the form:

```
DECLARE TERMINAL$ATTRIBUTES STRUCTURE(
     NUM$WORDS           WORD,
     NUM$USED            WORD,
     CONNECTION$FLAGS    WORD,
     TERMINAL$FLAGS      WORD,
     IN$BAUD$RATE        WORD,
     OUT$BAUD$RATE       WORD,
     SCROLL$LINES        WORD,
     X$Y$SIZE            WORD,
     X$Y$OFFSET          WORD,
     FLOW$CONTROL        WORD,
     HIGH$WATER$MARK     WORD,
     LOW$WATER$MARK      WORD,
     FC$ON$CHAR          WORD,
     FC$OFF$CHAR         WORD);
```

where the fields are defined in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL. If you are using the Terminal Support Code, this special subfunction is invisible to the terminal device driver.

In a request to set up special character recognition in the input stream of a terminal driver for signalling purposes, FUNCT equals special, SUBFUNCT equals signal, and AUX$P points to a structure of the form:

```
DECLARE SIGNAL$CHARACTER STRUCTURE(
     SEMAPHORE           SELECTOR
     CHARACTER           BYTE);
```

where the fields are defined in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL. In a request to read a tape file mark, FUNCT equals special, SUBFUNCT equals read tape file mark, and AUX$P points to a structure of the form:

```
DECLARE READ$FILE$MARK STRUCTURE(
     SEARCH              BYTE);
```

where the field is defined in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

LINK$FOR	POINTER that the device driver/device driver support routines can use to implement a request queue. This field points to the location of the next IORS in the queue.
LINK$BACK	POINTER that the device driver/device driver support routines can use to implement a request queue. This field points to the location of the previous IORS in the queue.

RESP$MBOX	SELECTOR that the I/O System fills with either an iRMX 86 token for the response mailbox or the address of an iRMX 88 exchange. Upon completion of the I/O request, the device driver/device driver support routines must send the IORS to this response mailbox or exchange.
DONE	BYTE that the device driver can set to TRUE (0FFH) or FALSE (00H) to indicate whether the entire request has been completed.
FILL	Reserved BYTE.
CANCEL$ID	SELECTOR used to identify queued I/O requests that CANCEL$IO can remove from the queue.
CONN$T	SELECTOR used in requests to the iRMX 86 I/O System. This field contains the token of the iRMX 86 file connection through which the request was issued.

DEVICE INTERFACES

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

***

CHAPTER 3 CATEGORIES AND PROPERTIES OF DEVICES AND DRIVERS

There are four types of device drivers in the iRMX 86 environment: common, random access, custom, and terminal. There are three types of device drivers in the iRMX 88 environment: common, random access, and custom. This chapter defines the distinctions between the types of drivers and discusses the characteristics and data structures pertaining to common and random access device drivers. Chapters 5, 6, and 7 are devoted to explaining how to write the various types of device drivers.

CATEGORIES OF DEVICES

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

COMMON DEVICES

Common devices are relatively simple devices other than terminals, such as line printers. This category includes devices that conform to the following conditions:

- A first-in/first-out mechanism for queuing requests is sufficient for accessing these devices.
- Only one interrupt level is needed to service a device.
- Data either read or written by these devices does not need to be broken up into blocks.

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

RANDOM ACCESS DEVICES

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

- A first-in/first-out mechanism for queuing requests is sufficient for accessing these devices.
- Only one interrupt level is needed to service the device.
- I/O requests must be broken up into blocks of a specific length.
- The device supports random access seek operations.

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

TERMINAL DEVICES

A terminal device is characterized by the fact that it reads and writes single characters, with an interrupt for each character. Because such devices are entirely different than common, random access, and even custom devices, terminal drivers and their required data structures are described in Chapter 7. The remainder of this chapter applies only to common, random access, and custom device drivers.

CUSTOM DEVICES

If your device fits neither the common nor the random access category, and is not a terminal or terminal-like device, you must write the entire driver for the device. The requirements of a custom device driver are defined in Chapter 6.

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

The I/O System supplies the common and random access routines that it calls when processing I/O requests. Flow charts for these procedures appear in Appendix A. The names and functions of these procedures are as follows: (The "RAD$" prefix applies to iRMX 88 routine names.)

Routine	Function
(RAD$)INIT$IO	Creates the resources needed by the remainder of the driver routines, creates an interrupt task, and calls a user-supplied routine that initializes the device itself.
(RAD$)FINISH$IO	Deletes the resources used by the other driver routines, deletes the interrupt task, and calls a user-supplied procedure that performs final processing on the device itself.

Routine	Function
(RAD$)QUEUE$IO	Places I/O requests (IORSs) on the queue of requests.
(RAD$)CANCEL$IO	Removes one or more requests from the request queue, possibly stopping the processing of a request that has already been started.

These routines process I/O requests for both common and random access devices. They distinguish between categories based on the value of the NUM$BUFFERS field in the unit's device-unit information block (DUIB). (When calling each of these routines, the I/O System supplies a pointer to the DUIB as one of the parameters.) If the NUM$BUFFERS field is nonzero, the routines assume the device is a random access device. If the NUM$BUFFERS field is zero, the routines assume the device is a common device.

In addition to the routines just described, the I/O System supplies an interrupt handler (interrupt service routine) and an interrupt task (called INTERRUPT$TASK) which respond to all interrupts generated by the units of a device, process those interrupts, and start the device working on the next I/O request on the queue. The INIT$IO procedure creates the interrupt task.

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

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

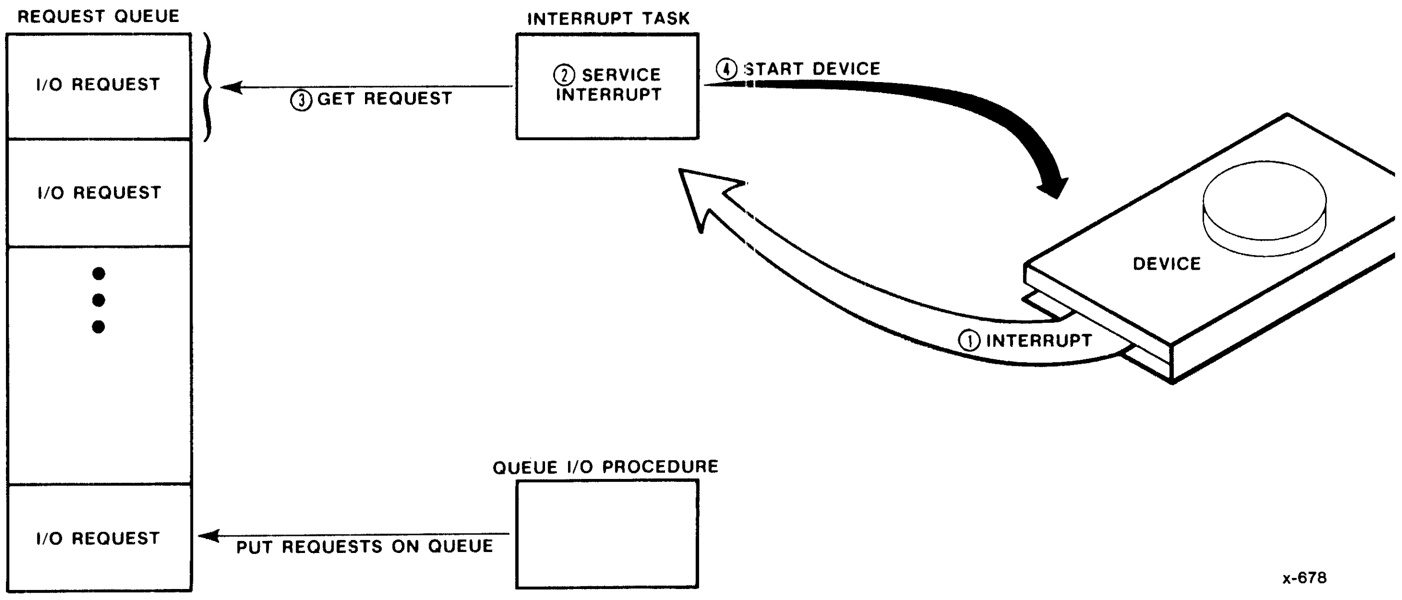

Figure 3-1. Interrupt Task Interaction

I/O SYSTEM ALGORITHM FOR CALLING THE DEVICE DRIVER PROCEDURES

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

1. To start I/O processing, an application task must make an I/O request. It can do this by invoking any of a variety of system calls. However, if you are an iRMX 86 user, the first I/O request to each device-unit must be an RQ$A$PHYSICAL$ATTACH$DEVICE system call, and if you are an iRMX 88 user, the first request to each device-unit must be either a DQ$ATTACH or a DQ$CREATE system call.

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

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

4. If you are an iRMX 86 user and the request just queued resulted from an iRMX 86 RQ$A$PHYSICAL$DETACH$DEVICE system call, the I/O System checks to see if any other units of the device are currently attached. If no other units of the device are attached, the I/O System calls the Finish I/O procedure to do any final processing on the device and clean up resources used by the device driver routines.

   If you are an iRMX 88 user and the request just queued resulted from either a DQ$DETACH or a DQ$DELETE system call, the I/O System checks to see if any other units of the device are currently attached. If no other units of the device are attached, the I/O System calls the Finish I/O procedure to do any final processing on the device and clean up resources used by the device driver routines.

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

- If the user makes an RQ$A$PHYSICAL$DETACH$DEVICE system call specifying the hard detach option, to forcibly detach the connection objects associated with a device-unit. The iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL describes the hard detach option.

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

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

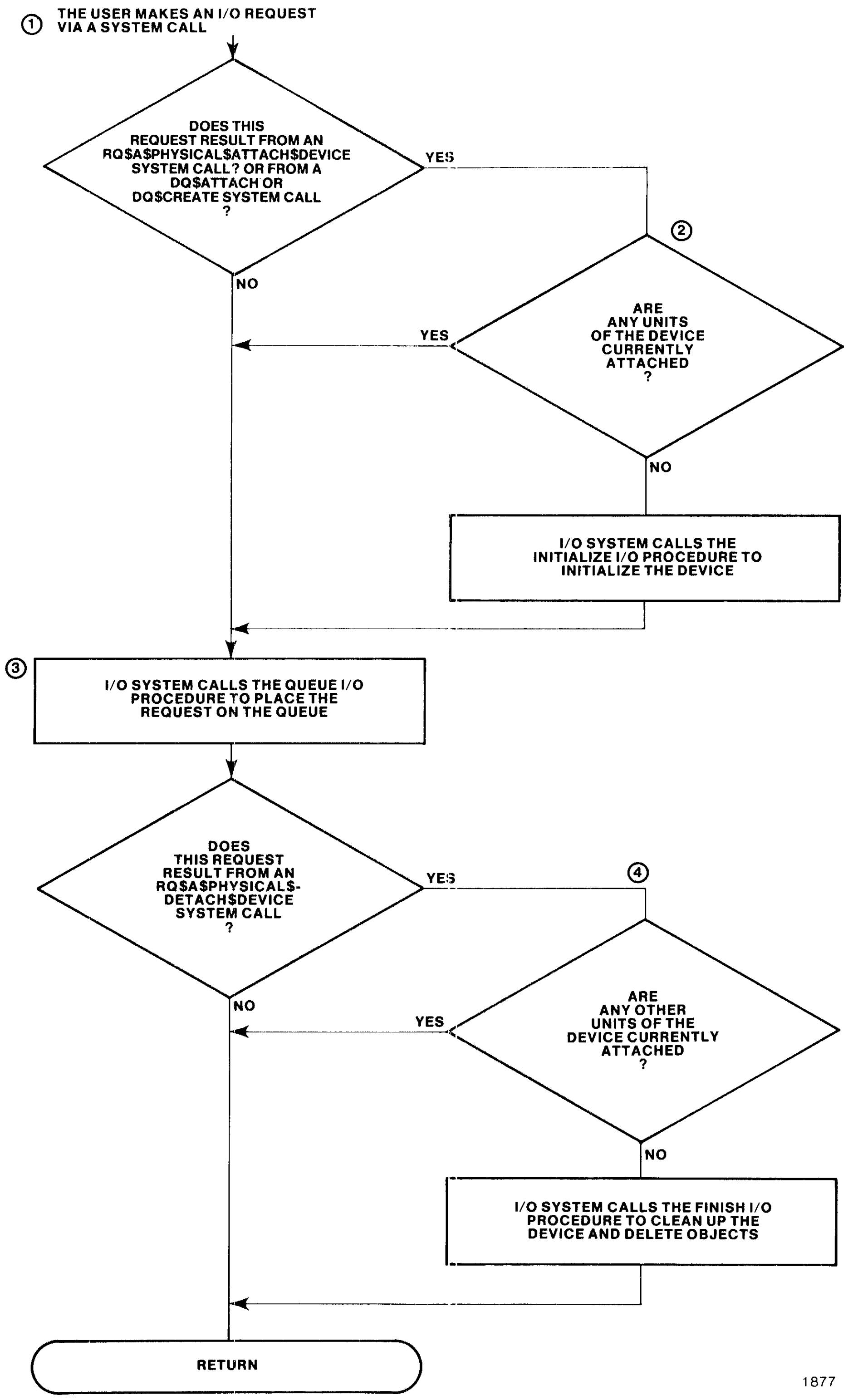
① THE USER MAKES AN I/O REQUEST VIA A SYSTEM CALL
DOES THIS REQUEST RESULT FROM AN RQ$A$PHYSICAL$ATTACH$DEVICE SYSTEM CALL? OR FROM A DQ$ATTACH OR DQ$CREATE SYSTEM CALL ?
YES
NO
②
ARE ANY UNITS OF THE DEVICE CURRENTLY ATTACHED ?
YES
NO
I/O SYSTEM CALLS THE INITIALIZE I/O PROCEDURE TO INITIALIZE THE DEVICE
③ I/O SYSTEM CALLS THE QUEUE I/O PROCEDURE TO PLACE THE REQUEST ON THE QUEUE
DOES THIS REQUEST RESULT FROM AN RQAPHYSICAL$-DETACH$DEVICE SYSTEM CALL ?
YES
NO
④
ARE ANY OTHER UNITS OF THE DEVICE CURRENTLY ATTACHED ?
YES
NO
I/O SYSTEM CALLS THE FINISH I/O PROCEDURE TO CLEAN UP THE DEVICE AND DELETE OBJECTS
RETURN
1877

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

REQUIRED DATA STRUCTURES

In order for the I/O System-supplied routines to be able to call the user-supplied routines, you must supply the addresses of these user-supplied routines, as well as other information, in a Device Information Table. In addition, processing I/O requests through a random access driver requires a Unit Information Table. Each DUIB contains one pointer field for a Device Information Table and another for a Unit Information Table.

DUIBs that correspond to units of the same device should point to the same Device Information Table, but they can point to different Unit Information Tables, if the units have different characteristics. Figure 3-3 illustrates this.

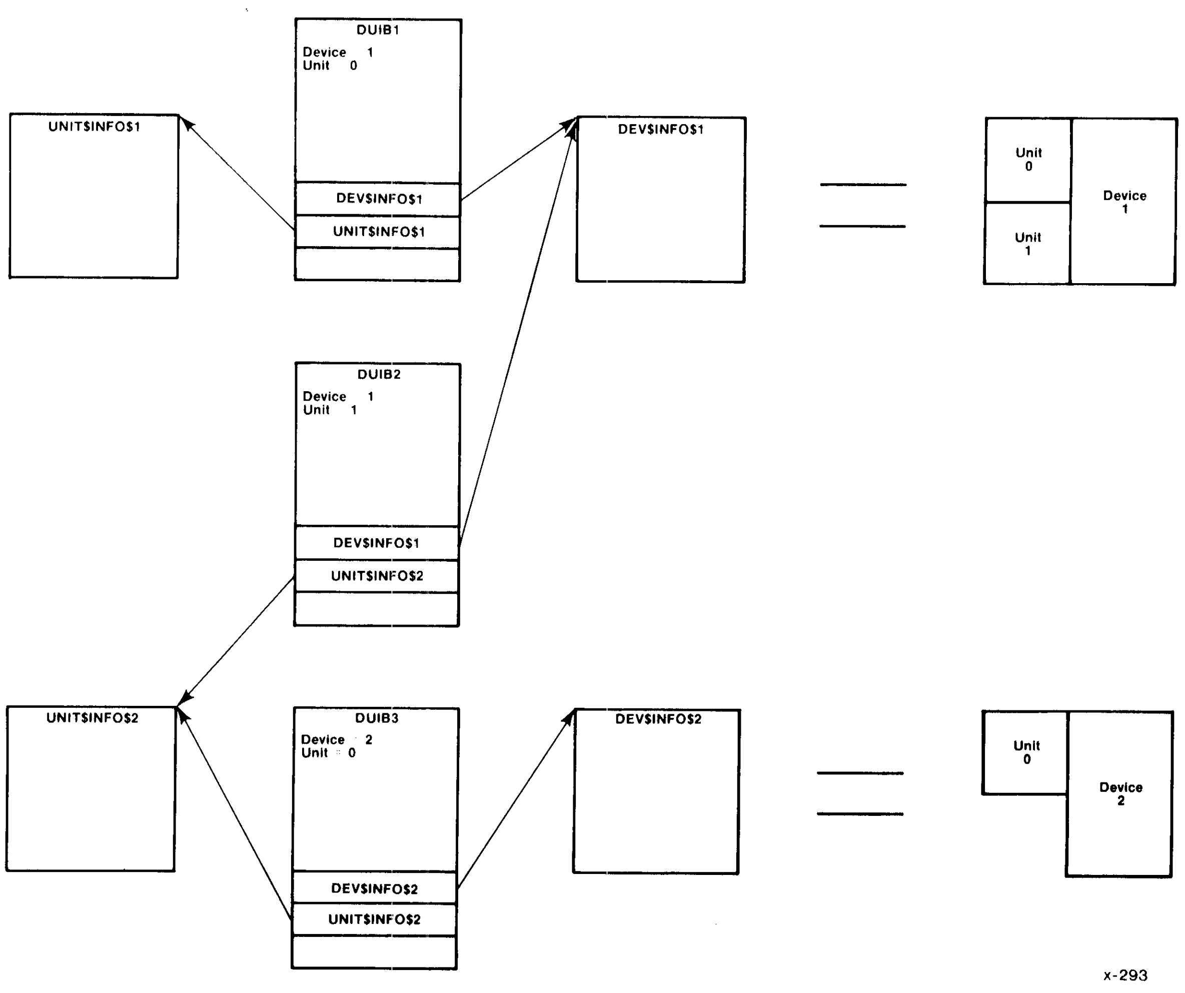

Figure 3-3. DUIBs, Device and Unit Information Tables

DEVICE INFORMATION TABLE

Common and random access Device Information Tables contain the same fields in the same order. When creating Device Information Tables for iRMX 86 applications, code them in the format shown here (as assembly-language structures). If you give the iRMX 86 ICU the pathname of your Unit Information Table file, the ICU includes the file in the assembly of IDEVCF.A86 (a Basic I/O System configuration file). IDEVCF.A86 contains the definition of the structure.

The fields DEVICE$INIT, DEVICE$FINISH, DEVICE$START, DEVICE$STOP, and DEVICE$INTERRUPT contain the names of user-supplied procedures whose duties are described in Chapter 5. When creating the file containing your Device Information Tables, specify external declarations for these user-supplied procedures. This allows the code for these user-supplied procedures to be included into the assembly of the I/O System. For example, if your procedures are named DEVICE$INIT, DEVICE$FINISH, DEVICE$START, DEVICE$STOP, and DEVICE$INTERRUPT, include the following declarations in the file containing your Device Information Tables:

```
extrn device$init: near
extrn device$finish: near
extrn device$start: near
extrn device$stop: near
extrn device$interrupt: near
```

The iRMX 88 ICU prompts you for each field in the Device Information Table structure. The iRMX 88 ICU generates the Device Information Table and places it in the device configuration source file.

Use the following format when coding your Device Information Tables:

```
RADEV_DEV_INFO  <
&    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 to associate itself with the correct interrupt level. The values for this field are encoded as follows:

iRMX 86 VALUES

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.

iRMX 88 VALUES

The values available are 0 through 3FH. Refer to the iRMX 88 REFERENCE MANUAL for further information.

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

DEVICE$INIT	WORD specifying the start address of a user-written device initialization procedure. The format of this procedure, which INIT$IO calls, is described in Chapter 5.
DEVICE$FINISH	WORD specifying the start address of a user-written device finish procedure. The format of this procedure, which FINISH$IO calls, is described in Chapter 5.
DEVICE$START	WORD specifying the start address of a user-written device start procedure. The format of this procedure, which QUEUE$IO and INTERRUPT$TASK call, is described in Chapter 5.
DEVICE$STOP	WORD specifying the start address of a user-written device stop procedure. The format of this procedure, which CANCEL$IO calls, is described in Chapter 5.
DEVICE$INTERRUPT	WORD specifying the start address of a user-written device interrupt procedure. The format of this procedure, which INTERRUPT$TASK calls, is described in Chapter 5.

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

UNIT INFORMATION TABLE

If you have random access device drivers in your system, you must create a Unit Information Table for each different type of unit in your system. Each random access device-unit's DUIB must point to one Unit Information Table, although multiple DUIBs can point to the same Unit Information Table. The Unit Information Table must include all information that is unit-dependent.

When creating Unit Information Tables for iRMX 86 applications, code them in the format shown here (as assembly-language structures). If you give the iRMX 86 ICU the pathname of your Unit Information Table file, the ICU includes the file in the assembly of IDEVCF.A86 (a Basic I/O System configuration file). IDEVCF.A86 contains the definition of the structure.

The iRMX 88 ICU prompts you for some fields in the Unit Information Table structure. The iRMX 88 ICU generates the Unit Information Table and places it in the device configuration source file.

The minimum requirements for the structure of the Unit Information Table are as follows:

```
RADEV_UNIT_INFO  <
&    TRACK$SIZE,          ; word
&    MAX$RETRY,           ; word
&    CYLINDER$SIZE        ; word
&    >
```

where:

TRACK$SIZE	WORD specifying the size, in bytes, of a single track of a volume on the unit. If the device controller supports reading and writing across track boundaries, and your driver is a random-access driver, place a zero in this field. If you specify a zero for this field, the I/O System-supplied random access support procedures place an absolute sector number in the DEV$LOC field of the IORS. If you specify a nonzero value for this field, the random access support 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$LOC field of the IORS and the track number in the high-order word of the DEV$LOC field before calling a user-written device start procedure. Instructions for writing a device start procedure are contained in Chapter 5.
MAX$RETRY	WORD specifying the maximum number of times an I/O request should be tried if an error occurs. Nine is the recommended value for this field. When this field contains a nonzero value, the I/O System-supplied procedures guarantee that read or write requests are retried if the user-supplied device start or device interrupt procedures return an IO$SOFT condition in the IORS.UNIT$STATUS field. (The IORS.UNIT$STATUS field is described in the "IORS Structure" section of Chapter 2.)
CYLINDER$SIZE	For iRMX 86 systems, a WORD whose meaning depends on its value, as follows:

0	The I/O System never requests a seek operation. Instead, it expects the device driver/controller to perform implied "seeks" when a read/write on the unit begins on a cylinder which is different from the one associated with the current position of the read/write head.
1	The I/O System automatically requests a seek operation (to seek to the correct cylinder) before performing a read or write. The device driver for the unit must call the SEEK$COMPLETE procedure immediately following each seek operation.

Other	Any other value specifies the number of sectors in a cylinder on the unit. The I/O System automatically requests a seek operation whenever a requested read or write operation on the unit begins in a different cylinder than that associated with the current position of the read/write head. The device driver for the unit must call the SEEK$COMPLETE procedure immediately following each seek operation.

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

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

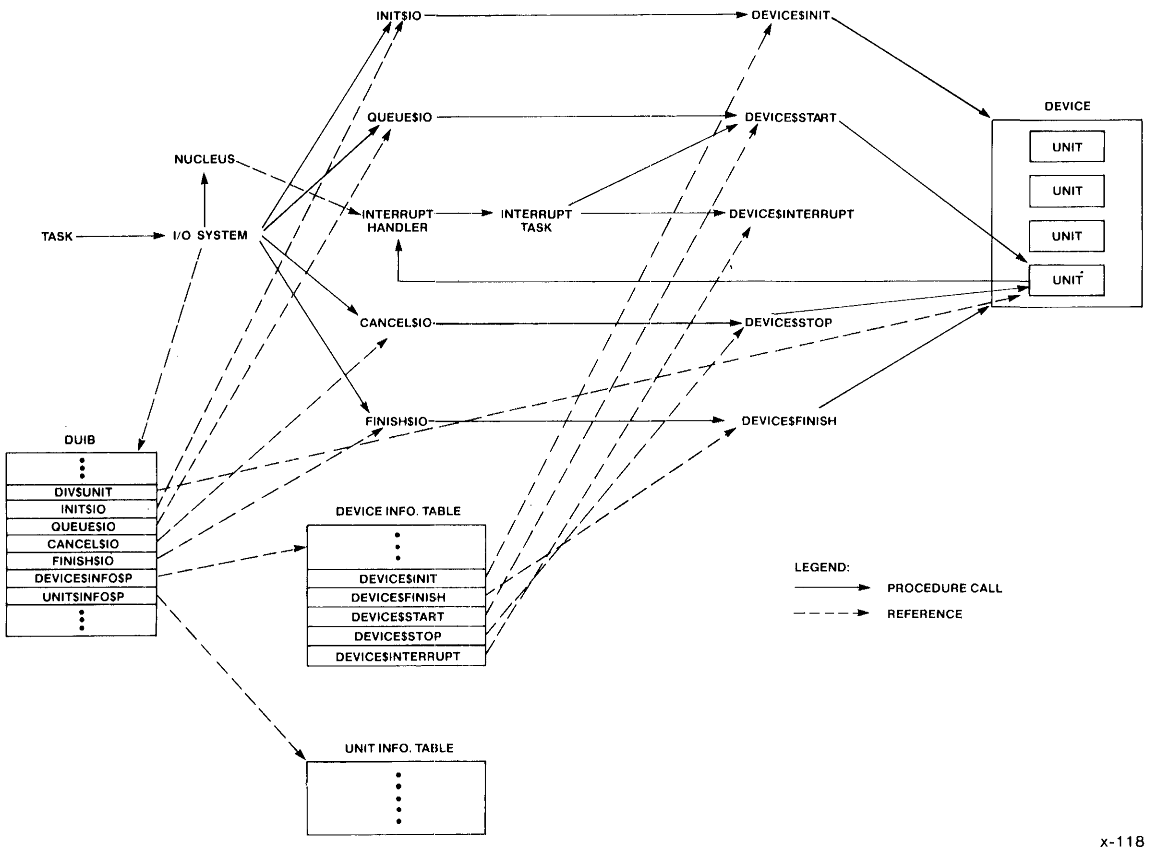

Figure 3-4. Relationships Between I/O Procedures and I/O Data Structures

DEVICE DATA STORAGE AREA

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

First, all device driver procedures that service individual units of the device can access and update the same data. The Initialize I/O procedure passes the address of the area back to the I/O System, which in turn gives the address to the other procedures of the driver.

They can then place information relevant to the device as a whole into the area. The identity of the first IORS on the request queue is maintained in this area, as well as the attachment status of the individual units and a means of accessing the interrupt task.

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

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

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

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

***

CHAPTER 4 I/O REQUESTS

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

I/O SYSTEM RESPONSES TO I/O REQUESTS

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

ATTACH DEVICE REQUESTS

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

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

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

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

DETACH DEVICE REQUESTS

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

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

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

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

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

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

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

CANCEL REQUESTS

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

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

DUIB AND IORS FIELDS USED BY DEVICE DRIVERS

Tables 4-1, 4-2, and 4-3 indicate, for each type of device driver, the fields of DUIBs and IORSs with which user-written portions of device drivers need to be concerned.

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

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

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

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

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

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

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

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

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

***

CHAPTER 5 WRITING COMMON OR RANDOM ACCESS DEVICE DRIVERS

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

In addition to providing information about the procedures that common or random access drivers must supply, this chapter describes the purpose and calling sequence for each of five procedures, two of which random access device drivers in iRMX 86 applications must call under certain conditions.

INTRODUCTION TO PROCEDURES THAT DEVICE DRIVERS MUST SUPPLY

The routines that are provided by the I/O System and that the I/O System calls (INIT$IO, FINISH$IO, QUEUE$IO, CANCEL$IO, and INTERRUPT$TASK for iRMX 86 systems) (RAD$INIT$IO, RAD$FINISH$IO, RAD$QUEUE$IO, RAD$CANCEL$IO, and INTERRRUPT$TASK for iRMX 88 systems) constitute the bulk of a common or random access device driver. These routines, in turn, make calls to device-dependent routines that you must supply. These device-dependent routines are described here briefly and then are presented in detail:

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

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

<u>A device start procedure.</u> 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.

<u>A device stop procedure.</u> 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.

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

DEVICE INITIALIZATION PROCEDURE

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

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

where:

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

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 to perform final processing on the device, after the last I/O request has been processed. The format of the call to the device finish procedure is as follows:

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

where:

device$finish	Name of the device finish procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
duib$p	POINTER to the DUIB of the device-unit being detached. From this DUIB, the device finish procedure can obtain the Device Information Table, where information such as the I/O port address is stored.
ddata$p	POINTER to the user portion of the device's data storage area. The device finish procedure should obtain, from this data area, identification of any resources other user-written procedures may have created, and delete these resources.

If you have a device that does not require any final processing, you can use the default device finish procedure supplied by the I/O System. The name of this procedure is DEFAULT$FINISH. Specify this name in the Device Information Table. DEFAULT$FINISH merely returns control to the caller. It is normally used when the default initialization procedure DEFAULT$INIT is used.

DEVICE START PROCEDURE

Both QUEUE$IO and INTERRUPT$TASK make calls to the device start procedure to start an I/O function. QUEUE$IO calls this procedure on receiving an I/O request when the request queue is empty. INTERRUPT$TASK calls the device start procedure after it finishes one I/O request if there are one or 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 to obtain information such as the type of I/O function requested, the address on the device of the byte where I/O is to commence, and the buffer address.

duib$p	POINTER to the DUIB of the device-unit for which the I/O request is intended. The device start procedure can use the DUIB to access the Device Information Table, where information such as the I/O port address is stored.
ddata$p	POINTER to the user portion of the device's data storage area. The device start procedure can use this data area to set flags or store data.

The device start procedure must do the following:

- It must be able to start the device processing any of the functions supported by the device and recognize that requests for nonsupported functions are error conditions.
- If it transfers any data, it must update the IORS.ACTUAL field to reflect the total number of bytes of data transferred (that is, if it transfers 128 bytes of data, it must put 128 in the IORS.ACTUAL field).
- If an error occurs when the device start procedure tries to start the device (such as on an write request to a write-protected disk), the device start procedure must set the IORS.STATUS field to indicate an E$IO condition and the IORS.UNIT$STATUS field to a nonzero value. The lower four bits of the field should be set as indicated in the "IORS Structure" section of Chapter 2. The remaining bits of the field can be set to any value (for example, the iSBC 204 device driver returns the device's result byte in the remainder of this field). If the function completes without an error, the device start procedure must set the IORS.STATUS field to indicate an E$OK condition.
- If the device start procedure determines that the I/O request has been processed completely, either because of an error or because the request has completed successfully, it must set the IORS.DONE field to TRUE. The I/O request will not always be completed; it may take several calls to the device interrupt procedure before a request is completed. However, if the request is finished and the device start procedure does not set the IORS.DONE field to TRUE, the device driver support routines wait until the device sends an interrupt and the device interrupt procedure sets IORS.DONE to TRUE, before determining that the request is actually finished.

DEVICE STOP PROCEDURE

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

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

where:

device$stop	Name of the device stop procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include this name in the Device Information Table.
iors$p	POINTER to the IORS of the request. The device stop procedure needs this information to determine what type of function to stop.
duib$p	POINTER to the DUIB of the device-unit on which the I/O function is being performed.
ddata$p	POINTER to the user portion of the device's data storage area. The device stop procedure can use this area to store data, if necessary.

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

DEVICE INTERRUPT PROCEDURE

INTERRUPT$TASK calls the user-written device interrupt procedure to process an interrupt that just occurred. 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 either that there are no requests on the request queue and the interrupt is extraneous or that the unit is completing a seek or other long-term operation.
duib$p	POINTER to the DUIB of the device-unit on which the I/O function was performed.

ddata$p	POINTER to the user portion of the device's data storage area. The device interrupt procedure can update flags in this data area or retrieve data sent by the device.

The device interrupt procedure must do the following:

- It must determine whether the interrupt resulted from the completion of an I/O function by the correct device-unit.

- If the correct device-unit did send the interrupt, the device interrupt procedure must determine whether the request is

  finished. If the request is finished, the device interrupt procedure must set the IORS.DONE field to TRUE.

- It must process the interrupt. This may involve setting flags in the user portion of the data storage area, tranferring data written by the device to a buffer, or some other operation.

- If an error has occurred, it must set the IORS.STATUS field to indicate an E$IO condition and the IORS.UNIT$STATUS field to a nonzero value. The lower four bits of the field should be set as indicated in the "IORS Structure" section of Chapter 2. The remaining bits of the field can be set to any value (for example, the iSBC 204 and 206 device drivers return the device's result byte in the remainder of this field). It must also set the IORS.DONE field to TRUE, indicating that the request is finished because of the error.

- If no error has occurred, it must set the IORS.STATUS field to indicate an E$OK condition.

PROCEDURES THAT iRMX™ 86 RANDOM ACCESS DRIVERS MUST CALL

There are several procedures that random access drivers in iRMX 86 applications can call under certain well-defined circumstances. They are NOTIFY, SEEK$COMPLETE, and procedures for the long-term operations (BEGIN$LONG$TERM$OP, END$LONG$TERM$OP, and GET$IORS).

NOTIFY PROCEDURE

Whenever a door to a flexible diskette drive is opened or the STOP button on a hard disk drive is pressed, the device driver for that device must notify the I/O System that the device is no longer available. The device driver does this by calling the NOTIFY procedure. When called in this manner, the I/O System stops accepting I/O requests for files on that device unit. Before the device unit can again be available for I/O requests, the application must detach it by a call to A$PHYSICAL$DETACH$DEVICE and reattach it by a call to A$PHYSICAL$ATTACH$DEVICE. Moreover, the application must obtain new file connections for files on the device unit.

In addition to not accepting I/O requests for files on that device unit, the I/O System will respond by sending an object to a mailbox. For this to happen, however, the object and the mailbox must have been established for this purpose by a prior call to A$SPECIAL, with the spec$func argument equal to FS$NOTIFY (2). (The A$SPECIAL system call is described in the BASIC I/O SYSTEM REFERENCE MANUAL.) The task that awaits the object at the mailbox has the responsibility of detaching and reattaching the device unit and of creating new file connections for files on the device unit.

The syntax of the NOTIFY procedure is as follows:

```
    CALL NOTIFY(unit, ddata$p);
```

where:

unit	BYTE containing the unit number of the unit on the device that went off-line.
ddata$p	POINTER to the user portion of the device's data storage area. This is the same pointer that is passed to the device driver by way of either the device$start or the device$interrupt procedure.

SEEK$COMPLETE PROCEDURE

In most applications, it is desirable to overlap seek operations (which can take relatively long periods of time) with other operations. To facilitate this, a device driver receiving a seek request can take the following actions in the following order:

1. The device start procedure starts the requested seek operation.

2. Depending on the kind of device, either the device start procedure or the device interrupt procedure sets the DONE flag in the IORS to TRUE (0FFH).

   - Some devices send only one interrupt in response to a seek request -- the one that indicates the completion of the seek. If your device operates in this manner, the device start procedure sets the DONE flag to TRUE (0FFH) immediately.

   - Some devices send two interrupts in response to a seek request -- one upon receipt of the request and one upon completion of the seek. If your device operates in this manner, the device start procedure leaves the DONE flag in the IORS set to FALSE (0).

     When the first interrupt from the device arrives, the device interrupt procedure sets the DONE flag to TRUE (0FFH).

3. When the interrupt from the device arrives (the one that indicates the completion of the seek), the device interrupt procedure calls the SEEK$COMPLETE procedure to signal the completion of the seek operation.

This process enables the device driver to handle I/O requests for other units on the device while the seek is in progress, thereby increasing the performance of the I/O System.

The syntax of the call to SEEK$COMPLETE is as follows:

```
    CALL SEEK$COMPLETE(unit, ddata$p);
```

where:

unit	BYTE containing the number of the unit on the device on which the seek operation is completed.
ddata$p	POINTER to the user portion of the device's data storage area. This is the same pointer that the random access support routines passes to the device start and device interrupt procedures.

Note that if your device driver calls the SEEK$COMPLETE procedure when a seek operation is completed, the CYLINDER$SIZE field of the Unit Information Table for the device unit should be configured greater than zero. On the other hand, if the driver does not call SEEK$COMPLETE, then CYLINDER$SIZE must be configured to zero.

PROCEDURES FOR OTHER LONG-TERM OPERATIONS

The iRMX 86 Operating System provides three procedures which device drivers can use to overlap long-term operations (such as tape rewinds) with other I/O operations. The procedures are BEGIN$LONG$TERM$OP, END$LONG$TERM$OP, and GET$IORS. These procedures are intended specifically for use with devices that do not support seek operations (such as tape drives).

BEGIN$LONG$TERM$OP Procedure

The BEGIN$LONG$TERM$OP procedure informs the random access support routines that a long-term operation is in progress, and that the support routines do not have to wait for the operation to complete before servicing other units on the device. Calling BEGIN$LONG$TERM$OP allows the controller to service read and write requests on other units of the device while the long-term operation is in progress.

To use BEGIN$LONG$TERM$OP, the device driver receiving the request for the long-term operation should take the following actions:

1. The device start procedure starts the long-term operation.

2. Depending on the kind of device, either the device start procedure or the device interrupt procedure sets the DONE flag in the IORS to TRUE (0FFH).

   - Some devices send only one interrupt in response to a request for a long-term operation -- the one that indicates the completion of the operation. If your device operates in this

     manner, the device start procedure sets the DONE flag to TRUE (0FFH) immediately.

   - Some devices send two interrupts in response to a request for a long-term operation -- one upon receipt of the request and one upon completion of the operation. If your device operates in this manner, the device start procedure leaves the DONE flag in the IORS set to FALSE (0). When the first interrupt from the device arrives, the device interrupt procedure sets the DONE flag to TRUE (0FFH).

3. The procedure that just set the DONE flag to TRUE (either the device start or device interrupt procedure) calls BEGIN$LONG$TERM$OP.

The syntax of the call to BEGIN$LONG$TERM$OP is as follows:

```
CALL BEGIN$LONG$TERM$OP(unit, ddata$p);
```

where:

unit	BYTE containing the number of the unit on the device which is performing the long-term operation.
ddata$p	POINTER to the user portion of the device's data storage area. This is the same pointer that the random access support routines passes to the device start and device interrupt procedures.

If your driver calls BEGIN$LONG$TERM$OP, it must also call END$LONG$TERM$OP when the device sends an interrupt to indicate the end of the long-term operation.

END$LONG$TERM$OP Procedure

The END$LONG$TERM$OP procedure informs the random access support routines that a long-term operation has completed. A driver that calls BEGIN$LONG$TERM$OP must also call END$LONG$TERM$OP or the driver cannot further access the unit that performed the long-term operation.

Specifically, when the unit sends an interrupt indicating the end of the long-term operation, the device interrupt procedure must call END$LONG$TERM$OP.

The syntax of the call to END$LONG$TERM$OP is as follows:

```
CALL END$LONG$TERM$OP(unit, ddata$p);
```

where:

unit	BYTE containing the number of the unit on the device which performed the long-term operation.
ddata$p	POINTER to the user portion of the device's data storage area. This is the same pointer that the random access support routines passes to the device start and device interrupt procedures.

GET$IORS Procedure

Long-term operations on some units involve multiple operations. For example, performing a rewind on some tape drives requires you to perform a rewind and a read file mark. The GET$IORS procedure allows your driver procedures to handle this situation without forcing you to write a custom driver for each device that is different.

GET$IORS allows your driver procedure to obtain the token of the IORS for the previous long-term request, so that it can modify the IORS to initiate new I/O requests. The IORS$P that INTERRUPT$TASK passed to the device interrupt procedure is set to zero (for units busy performing a seek or other long-term operation). Therefore, the driver can only access the IORS in this manner.

To use GET$IORS, the device driver performing the long-term operation should take the following actions:

1. The device driver starts the long-term operation and calls BEGIN$LONG$TERM$OP in the usual manner (as described in the "BEGIN$LONG$TERM$OP Procedure" section).

2. When the unit sends an interrupt indicating the end of the long-term operation, the device interrupt procedure calls GET$IORS to obtain the IORS.

3. The device interrupt procedure modifies the FUNCT and SUBFUNCT fields of the IORS to specify the next operation to perform. It also sets the DONE flag to FALSE (0).

4. The device interrupt procedure calls END$LONG$TERM$OPERATION.

The syntax of the call to GET$IORS is as follows:

```
iors$base = GET$IORS(unit, ddata$p);
```

where:

iors$base	SELECTOR in which the random access support routines return the base portion of the IORS. Use the PL/M-86 built-in procedure BUILD$PTR (specifying an offset of 0) to obtain a pointer to the IORS.
unit	BYTE containing the number of the unit on the device which performed the long-term operation.
ddata$p	POINTER to the user portion of the device's data storage area. This is the same pointer that the random access support routines passes to the device start and device interrupt procedures.

FORMATTING CONSIDERATIONS

If you write a random access driver and you intend to use the Human Interface FORMAT command (for iRMX 86 systems) or the RQ$FORMAT call (for iRMX 88 systems) to format volumes on that device, your driver routines must set the status field in the IORS in the manner that the FORMAT command expects.

When formatting volumes, the FORMAT command issues system calls (A$SPECIAL or S$SPECIAL) to format each track. It knows that formatting is complete when it receives an E$SPACE exception code in response. To be compatible with FORMAT, your driver must also return E$SPACE.

In particular, if your driver must perform some operation on the device to format it, your device interrupt procedure must set the IORS.STATUS to E$SPACE after the last track has been formatted.

However, if the device requires no physical formatting (for example, when formatting is a null operation for that device), your device start procedure can set IORS.STATUS to E$SPACE immediately after being called to start the formatting operation.

***

CHAPTER 6 WRITING A CUSTOM DEVICE DRIVER

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

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

A Finish I/O Procedure. This procedure must perform any final processing on the device and delete resources 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.

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

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

INITIALIZE I/O PROCEDURE

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

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

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

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

where:

init$io	Name of the Initialize I/O procedure. You can use any name for this procedure as long as it does not conflict with other procedure names. You must, however, provide its starting address in the DUIBs of all device-units that it services.
duib$p	POINTER to the DUIB of the device-unit for which the request is intended. The init$io procedure uses this DUIB to determine the characteristics of the unit.
ddata$p	POINTER to a WORD in which the init$io procedure can place the location of a data storage area, if the device driver needs such an area. If the device driver requires that a data area be associated with a device (to contain the head of the I/O queue, DUIB addresses, or status information), the init$io procedure should create this area and save its location via this pointer. If the driver does not need such a data area, the init$io procedure should return a zero via this pointer.
status$p	POINTER to a WORD in which the init$io procedure must place the status of the initialize operation. If the operation is completed successfully, the init$io procedure must return the E$OK condition code. Otherwise it should return the appropriate exception code. If the init$io procedure does not return the E$OK condition code, it must delete any resources that it has created.

FINISH I/O PROCEDURE

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

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

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

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

where:

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

QUEUE I/O PROCEDURE

The I/O System calls the Queue I/O procedure to place an I/O request on a queue, so that it can be processed when the device is not busy. The Queue I/O procedure must actually start the processing of the next I/O request on the queue 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$io	Name of the Queue I/O procedure. You can use any name for this procedure as long as it does not conflict with other procedure names. You must, however, provide its starting address for the DUIBs of all device-units that it services.
iors$t	SELECTOR containing the location of an IORS. This IORS describes the request. When the request is processed, the driver (though not necessarily the queue$io procedure) must fill in the status fields and send the IORS to the response mailbox (exchange) indicated in the IORS. Chapter 2 describes the format of the IORS. It lists the information that the I/O System supplies when it passes the IORS to the queue$io procedure and indicates the fields of the IORS that the device driver must fill in.

duib$p	POINTER to the DUIB of the device-unit for which the request is intended.
ddata$t	SELECTOR containing the location of the data storage area originally created by the init$io procedure. The queue$io procedure can place any necessary information in this area in order to update the request queue or status fields.

CANCEL I/O PROCEDURE

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

- If the user makes an RQ$A$PHYSICAL$DETACH$DEVICE system call and specifies the hard detach option (refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for a description of this call). This system call forcibly detaches all objects associated with a device-unit.
- If the job containing the task which made an I/O request is deleted. The I/O System calls the Cancel I/O procedure to remove any requests that tasks in the deleted job might have made.

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

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

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

where:

cancel$id	Name of the Cancel I/O procedure. You can use any name for this procedure as long as it doesn't conflict with other procedure names. You must, however, provide its starting address in the DUIBs of all device-units that it services.
cancel$id	WORD containing the id value for the I/O requests that are are to be cancelled. Any pending requests with this value in the cancel$id field of their IORS's must be removed from the queue of requests by the Cancel I/O procedure. Moreover, the I/O System places a CLOSE request with the same cancel$id value in the queue. The CLOSE request must not be processed until all other requests with that cancel$id value have been returned to the I/O System.

duib$p	POINTER to the DUIB of the device-unit for which the request cancellation is intended.
ddata$t	SELECTOR containing the location of the data storage area originally created by the init$io procedure. This area may contain the request queue.

IMPLEMENTING A REQUEST QUEUE

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

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

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

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

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

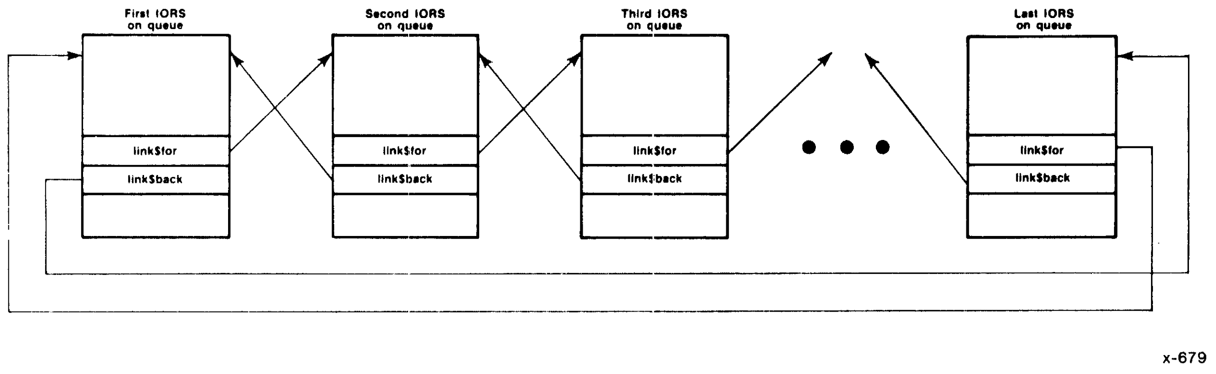

Figure 6-1. Request Queue

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

***

CHAPTER 7
TERMINAL DRIVERS

Both the iRMX 86 and iRMX 88 Operating Systems supply a Terminal Handler that can serve as an interface between the Nucleus and a terminal device. This interface is minimal and allows limited interaction between the terminal operator and the Operating System. However, the iRMX 86 Operating System also provides an interface to terminals via the Basic I/O System. This interface allows tasks to use the power and convenience of I/O System calls when communicating with terminals. To add support for new terminal controllers in the Basic I/O System, you can write device drivers, which provide the software link between the Operating System software (called the Terminal Support Code) and the terminal.

The iRMX 88 Executive does not support terminal drivers as outlined in this chapter.

This chapter explains how to write a terminal driver whose capabilities include handling single-character I/O, parity checking, answering and hanging up functions on a modem, and automatic baud rate searching for each of several terminals. Such a driver is neither common, random access, nor custom. Consequently, this chapter is more self-contained than Chapters 5 and 6; it describes the data structures used by terminal drivers, as well as the procedures that you must provide.

TERMINAL SUPPORT CODE

As in the case of common and random access drivers, the I/O System provides the procedures that the I/O System invokes when performing terminal I/O. They are known collectively as the Terminal Support Code. Figure 7-1 shows schematically the relationships between the various layers of code that are involved in driving a terminal.

Among the duties performed by the Terminal Support Code are managing buffers and maintaining several terminal-related modes.

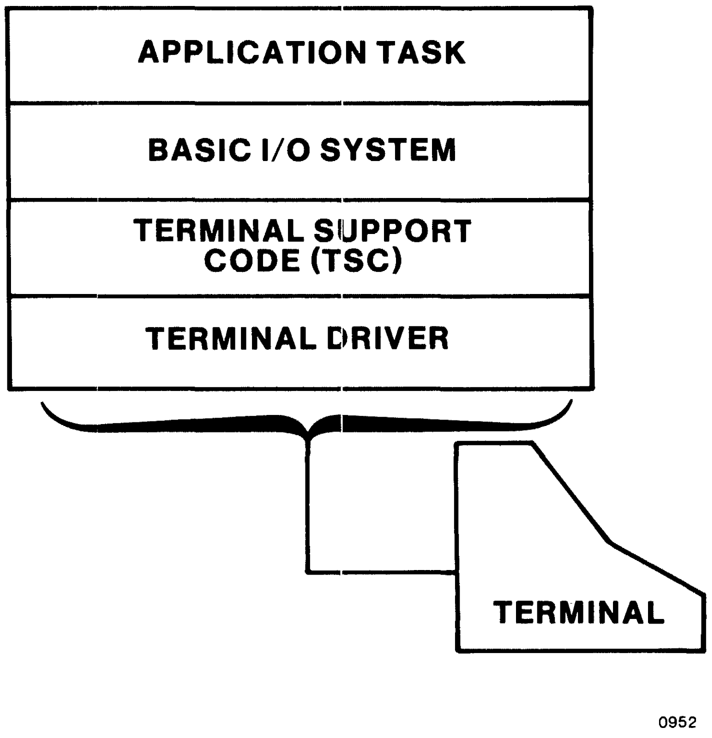

Figure 7-1. Software Layers Supporting Terminal I/O

DATA STRUCTURES SUPPORTING TERMINAL I/O

The principal data structures supporting terminal I/O are the Device-Unit Information Block (DUIB), Device Information Table, Unit Information Table, and the Terminal Support Code (TSC) data structure. These data structures are defined in the next few paragraphs.

DUIB

This section lists the elements that make up a DUIB for a device-unit that is a terminal. When creating DUIBs for iRMX 86 applications, code them in the format shown here (as assembly-language structures). If you give the iRMX 86 ICU the pathname of your Unit Information Table field, the iRMX 86 Interactive Configuration Utility (ICU) includes your DUIB file in the assembly of IDEVCF.A86 (a Basic I/O System configuration file). IDEVCF.A86 contains the definition of the structure.

```
DEFINE_DUIB  <
&    NAME,                  ; byte (14)
&    1,                     ; word - file$drivers - (physical)
&    OFBH,                  ; byte - functs - (no seek)
&    0,                     ; byte - flags - (not disk)
&    0,                     ; word - dev$gran - (not random access)
&    0,                     ; dword - dev$size - (not storage device)
&    DEVICE,                ; byte - (device dependent)
&    UNIT,                  ; byte - (unit dependent)
&    DEV$UNIT,              ; word - (device and unit dependent)
&    TSINITIO,              ; word - init$io - (terminal device)
&    TSFINISHIO,            ; word - finish$io - (terminal device)
&    TSQUEUEIO,             ; word - queue$io - (terminal device)
&    TSCANCELIO,            ; word - cancel$io - (terminal device)
&    DEVICE$INFO$P,         ; pointer - (address of
                            ; TERMINAL$DEVICE$INFO)
&    UNIT$INFO$P,           ; pointer - (address of
                            ; TERMINAL$UNIT$INFO)
&    OFFFFH,                ; word - update$timeout - (not disk)
&    0,                     ; word - num$buffers - (none)
&    PRIORITY,              ; byte - (I/O System dependent)
&    0,                     ; byte - fixed$update - (none)
&    0,                     ; byte - max$buffers - (none)
&    RESERVED,              ; byte
&    >
```

DEVICE INFORMATION TABLE

A terminal's Device Information Table provides information about a terminal controller. When creating these tables, code them in the format shown here (as assembly-language declarations). If you give the iRMX 86 ICU the pathname of your Unit Information Table field, the ICU includes the file in the assembly of IDEVCF.A86 (a Basic I/O System configuration file).

The fields TERM$INIT, TERM$FINISH, TERM$SETUP, TERM$OUT, TERM$ANSWER, TERM$HANGUP, and TERM$CHECK contain the names of user-supplied procedures whose duties are described later in this chapter. When creating the file containing your Device Information Tables, specify external declarations for these user-supplied procedures. This allows the code for these user-supplied procedures to be included in the generation of the I/O System. For example, if your procedures are named TERM$INIT, TERM$FINISH, TERM$SETUP, TERM$OUT, TERM$ANSWER, TERM$HANGUP, and TERM$CHECK, include the following declarations in the file containing your Device Information Tables:

```
    extrn term$init: near
    extrn term$finish: near
    extrn term$setup: near
    extrn term$out: near
    extrn term$answer: near
    extrn term$hangup: near
    extrn term$check: near
```

Use the following format when coding your Device Information Tables:

```
TERMINAL$DEVICE$INFORMATION
     DW   NUM$UNITS
     DW   DRIVER$DATA$SIZE
     DW   STACK$SIZE
     DW   TERM$INIT
     DW   TERM$FINISH
     DW   TERM$SETUP
     DW   TERM$OUT
     DW   TERM$ANSWER
     DW   TERM$HANGUP
     DW   NUM$INTERRUPTS
     INTERRUPTS
          DW   INTERRUPT$LEVEL
          DW   TERM$CHECK
               .                    ; define interrupt$level and
               .                    ; term$check for each interrupt
               .                    ; level
     DRIVER$INFO
          DB   DRIVER$INFO$1
          DB   DRIVER$INFO$2
               .
               .
               .
```

where:

NUM$UNITS	WORD containing the number of terminals on this terminal controller.
DRIVER$DATA$SIZE	WORD containing the number of bytes in the driver's data area pointed to by the USER$DATA$PTR field of the TSC Data structure.
STACK$SIZE	WORD containing the number of bytes of stack needed collectively by the user-supplied procedures in this device driver.
TERM$INIT	WORD specifying the address of this controller's user-written terminal initialization procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
TERM$FINISH	WORD specifying the address of this controller's user-written terminal finish procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
TERM$SETUP	WORD specifying the address of this controller's user-written terminal setup procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.

TERM$OUT	WORD specifying the address of this controller's user-written terminal output procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
TERM$ANSWER	WORD specifying the address of this controller's user-written terminal answer procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
TERM$HANGUP	WORD specifying the address of this controller's user-written terminal hangup procedure. When creating the Device Information Table, use the procedure name as a variable to supply this information.
NUM$INTERRUPTS	WORD containing the number of interrupt lines that this controller uses. You must define an INTERRUPT$LEVEL and TERM$CHECK word for each interrupt.
INTERRUPT$LEVEL	WORDs containing the level numbers of the interrupts that are associated with the terminals driven by this controller. You must supply one such word for each interrupt the controller uses.
TERM$CHECK	WORDs specifying the addresses of this controller's user-written terminal check procedures. Each TERM$CHECK field specifies the terminal check procedure for the INTERRUPT$LEVEL immediately preceding it. When creating the Device Information Table, use the procedure names as the variables to supply this information. If any of the TERM$CHECK words equals zero, there is no term$check procedure associated with the corresponding interrupt level. Instead, interrupts on these levels are assumed to be output ready interrupts which will cause TERM$OUT to be called.
DRIVER$INFO	BYTES or WORDS containing driver-dependent information.

NOTE

Usually, terminal drivers are concerned only with the DRIVER$INFO fields of the Device Information Table. Therefore, a terminal driver can declare a structure of the following form when accessing this data:

```
DECLARE
   TERMINAL$DEVICE$INFO STRUCTURE(
     FILLER(nbr$of$words)  WORD,
     DRIVER$INFO$1         BYTE,
     DRIVER$INFO$2         BYTE,
           .
           .
           .
     DRIVER$INFO$N         BYTE);
```

where nbr$of$words equals 10 + 2*(number of interrupt levels used by the driver)

You must supply the TERM$INIT, TERM$FINISH, TERM$SETUP, TERM$OUT, TERM$ANSWER, TERM$HANGUP, and TERM$CHECK procedures. However, if your terminals are not used with modems, the TERM$ANSWER and TERM$HANGUP procedures can simply contain a RETURN. Also, if your application does not need to perform special processing when all of the terminals on the controller are detached, the TERM$FINISH procedure also can simply contain a RETURN.

UNIT INFORMATION TABLE

A terminal's Unit Information Table provides information about an individual terminal. Although only one Device Information Table can exist for each driver (controller), several Unit Information Tables can exist if different terminals have different characteristics (such as baud rate, duplex, or parity, for example). When creating Unit Information Tables, code them in the format shown here (as assembly-language declarations). If you give the iRMX 86 ICU; the pathname of your Unit Information Table field, the ICU includes the file in the assemgly of IDEVCF.A86 (a Basic I/O System configuration file).

```
TERMINAL$UNIT$INFORMATION
     DW   CONN$FLAGS
     DW   TERM$FLAGS
     DW   IN$RATE
     DW   OUT$RATE
     DW   SCROLL$NUMBER
     DW   FLOW$CONTROL*
     DW   HIGH$WATER$MARK*
     DW   LOW$WATER$MARK*
     DW   FC$ON$CHAR*
     DW   FC$OFF$CHAR*
```

*These elements apply only to buffered device drivers and are useful only if you must specify them at configuration time.

where:

CONN$FLAGS — WORD specifying the default connection flags for this terminal. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information about these flags. The flags are encoded as follows. (Bit 0 is the low-order bit.)

Bits	Value and Meaning
0-1	Line editing control.
	0 = Invalid Entry.
	1 = No line editing (transparent mode).
	2 = Line editing (normal mode).
	3 = No line editing (flush mode).
2	Echo control.
	0 = Echo.
	1 = Do not echo.
3	Input parity control.
	0 = Set parity bit to 0.
	1 = Do not alter parity bit.
4	Output parity control.
	0 = Set parity bit to 0.
	1 = Do not alter parity bit.

Bits	Value and Meaning
5	Output control character control. 0 = Accept output control characters in the input stream. 1 = Ignore output control characters in the input stream.
6-7	OSC control sequence control. 0 = Act upon OSC sequences that appear in either the input or output stream. 1 = Act upon OSC sequences in the input stream only. 2 = Act upon OSC sequences in the output stream only. 3 = Do not act upon any OSC sequences.
8-15	Reserved bits. For future compatibility, set to 0.

TERM$FLAGS WORD specifying the terminal connection flags for this terminal. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information about these flags. The flags are encoded as follows. (Bit 0 is the low-order bit.)

Bits	Value and Meaning
0	Reserved bit. Set to 1.
1	Line protocol indicator. 0 = Full duplex. 1 = Half duplex.
2	Output medium. 0 = Video display terminal (VDT). 1 = Printed (Hard copy).
3	Modem indicator. 0 = Not used with a modem. 1 = Used with a modem.

Bits	Value and Meaning
4-5	Input parity control. 0 = Always set parity bit to 0. 1 = Never alter the parity bit. 2 = Even parity is expected on input. Set the parity bit to 0 unless the received byte has odd parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error.) 3 = Odd parity is expected in input. Set the parity bit to 0 unless the received byte has even parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error.)
6-8	Output parity control. 0 = Always set parity bit to 0. 1 = Always set parity bit to 1. 2 = Set parity bit to give the byte even parity. 3 = Set parity bit to give the byte odd parity. 4 = Do not alter the parity bit.
9	OSC Translation control. 0 = Do not enable translation. 1 = Enable translation.
10	Terminal axes sequence control. This specifies the order in which Cartesian-like coordinates of elements on a terminal's screen are to be listed or entered. 0 = List or enter the horizontal coordinate first.

Bits	Value and Meaning
	1 = List or enter the vertical coordinate first.
11	Horizontal axis orientation control. This specifies whether the coordinates on the terminal's horizontal axis increase or decrease as you move from left to right across the screen.
	0 = Coordinates increase from left to right.
	1 = Coordinates decrease from left to right.
12	Vertical axis orientation control. This specifies whether the coordinates on the terminal's vertical axis increase or decrease as you move from top to bottom across the screen.
	0 = Coordinates increase from top to bottom.
	1 = Coordinates decrease from top to bottom.
13-15	Reserved bits. For future compatibility, set to 0.

NOTE

If bits 4-5 contain 2 or 3, and bits 6-8 also contain 2 or 3, then they must both contain the same value. That is, they must both reflect the same parity convention (even or odd).

IN$RATE

WORD indicating the input baud rate. The word is encoded as follows:

0 =	Invalid.
1 =	Perform an automatic baud rate search.
Other =	Actual input baud rate, such as 9600.

OUT$RATE

WORD indicating the output baud rate. The word is encoded as follows:

0 =	Use the input baud rate for output.
Other =	Actual output baud rate, such as 9600.

	Most applications require the input and output baud rates to be equal. In such cases, use IN$RATE to set the baud rate and specify a zero for OUT$RATE.
SCROLL$NUMBER	WORD specifying the number of lines that are to be sent to the terminal each time the operator enters the appropriate control character (Control-W is the default).

The Unit Information Table can contain additional data, depending on the needs of the controller. Refer to the "Additional Information for Buffered Devices" section of this chapter for information about other fields you can add to the table.

TERMINAL SUPPORT CODE (TSC) DATA AREA

DUIBs, Device Information Tables, and Unit Information Tables are structures that you set up at configuration time to provide information about the initial state of your terminals. During configuration, the ICU assembles these tables into the code segment of the Basic I/O System. Therefore, they remain fixed throughout the life of the application system.

However, the Basic I/O System also provides a structure in the data segment (this section calls it the TSC Data Area) which changes to reflect the current state of the terminal controller and its units.

The TSC Data Area consists of three portions:

- A 30H-byte controller portion which contains information that applies to the device as a whole.
- A 400H-byte unit portion for each unit in the device. The NUM$UNITS field in the Device Information Table specifies the number of unit portions that the Basic I/O System creates.
- A user portion which the user-written driver routines can use in any manner they choose. The DRIVER$DATA$SIZE field in the Device Information Table specifies the length of this portion. One of the fields in the controller portion (USER$DATA$PTR) points to the beginning of this field.

Figure 7-2 illustrates the TSC Data Area graphically.

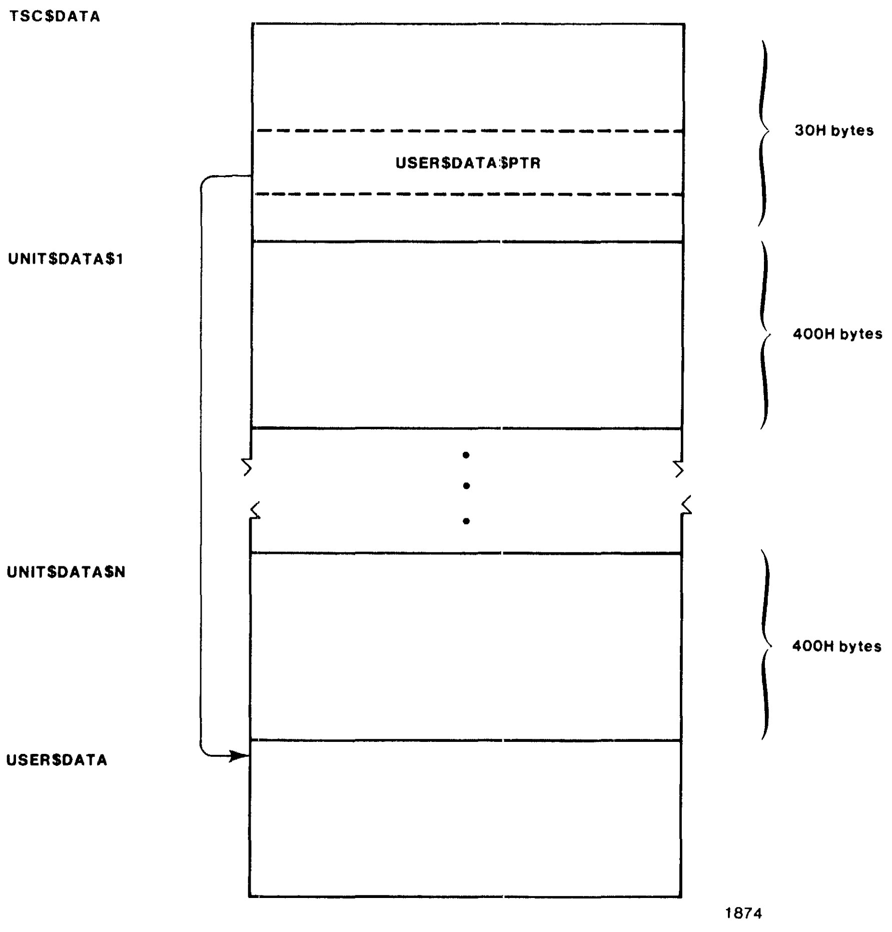

Figure 7-2. TSC Data Area

When the Basic I/O System calls one of your user-written driver procedures, it passes, as a parameter, a pointer either to the start of the TSC Data Area or to the start of one of the unit portions of the TSC Data Area. Your driver routines can then obtain information from the TSC Data Area or modify the information there.

The TSC Data Area always starts on a segment boundary Its structure is as follows:

```
DECLARE  TSC$DATA  STRUCTURE(
     IOS$DATA$SEGMENT                SELECTOR,
     STATUS                          WORD,
     INTERRUPT$TYPE                  BYTE,
     INTERRUPTING$UNIT               BYTE,
     DEV$INFO$PTR                    POINTER,
     USER$DATA$PTR                   POINTER,
     RESERVED(34)                    BYTE,
DECLARE  UNIT$DATA(*)  STRUCTURE(
     UNIT$INFO$PTR                   POINTER,
     TERMINAL$FLAGS                  WORD,
     IN$RATE                         WORD,
     OUT$RATE                        WORD,
     SCROLL$NUMBER                   WORD,
     RESERVED1(901)                  BYTE,
     BUFFERED$DEVICE$DATA(11)        BYTE,
     RESERVED2(100)                  BYTE);
```

where:

IOS$DATA$SEGMENT — SELECTOR containing the base address of the I/O System's data segment. The I/O System's terminal support routine TSINITIO fills in this information during initialization.

STATUS — WORD in which the user-written terminal initialization procedure must return status information.

INTERRUPT$TYPE — BYTE in which the user-written terminal check procedure must return the encoded interrupt type. The possible values are:

0	None
1	Input interrupt
2	Output interrupt
3	Ring interrupt
4	Carrier interrupt
5	Delay interrupt

If the terminal check procedure detects that there are more interrupts to service, the terminal check procedure adds the following value:

8	More interrupts

to the encoded interrupt type it returns.

For more information about these codes and their values, see the description of the terminal check procedure in the next section.

INTERRUPTING$UNIT — BYTE in which the user-written terminal check procedure must return the unit number of the interrupting device. This value identifies the unit that is interrupting.

DEV$INFO$PTR	POINTER to the Terminal Device Information Table for this controller. The I/O System's terminal support routine TSINITIO fills in this data during initialization.
USER$DATA$PTR	POINTER to the beginning of the user portion of the TSC Data Area. This user area can be used by the driver, as needed. The I/O System's terminal support routine TSINITIO fills in this pointer value during initialization.
UNIT$DATA	STRUCTUREs containing unit portions of the TSC Data Area. There is one structure for each unit (terminal) of the device. When a user attaches the unit (via the A$PHYSICAL$ATTACH$DEVICE system call or the ATTACHDEVICE Human Interface command, for example), the I/O System's terminal support routines initialize the appropriate UNIT$DATA structure. They perform the initialization by filling in all the fields of the UNIT$DATA structure with information from the DUIB and the Unit Information Table.
UNIT$INFO$PTR	POINTER to the Unit Information Table for this terminal. This is the same information as in the UNIT$INFO$P field of the DUIB for this device-unit (terminal).
TERMINAL$FLAGS	WORD specifying the connection flags for this terminal. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information about these flags. The flags are encoded as follows. (Bit 0 is the low-order bit.)

<u>Bits</u>	<u>Value and Meaning</u>
0	Reserved bit. Set to 1.
1	Line protocol indicator. 0 = Full duplex. 1 = Half duplex.
2	Output medium. 0 = Video display terminal (VDT). 1 = Printed (Hard copy).
3	Modem indicator. 0 = Not used with a modem. 1 = Used with a modem.

Bits	Value and Meaning
4-5	Input parity control.
	0 = Always set parity bit (bit 7) to 0.
	1 = Never alter the parity bit.
	2 = Even parity is expected on input. Set the parity bit to 0 unless the received byte has odd parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error.)
	3 = Odd parity is expected in input. Set the parity bit to 0 unless the received byte has even parity or there is some other error, such as (a) the received stop bit has a value of 0 (framing error) or (b) the previous character received has not yet been fully processed (overrun error.)
6-8	Output parity control.
	0 = Always set parity bit to 0.
	1 = Always set parity bit to 1.
	2 = Set parity bit to give the byte even parity.
	3 = Set parity bit to give the byte odd parity.
	4 = Do not alter the parity bit.
9	OSC Translation control.
	0 = Do not enable translation.
	1 = Enable translation.
10	Terminal axes sequence control. This specifies the order in which Cartesian-like coordinates of elements on a terminal's screen are to be listed or entered.
	0 = List or enter the horizontal coordinate first.

Bits	Value and Meaning
	1 = List or enter the vertical coordinate first.
11	Horizontal axis orientation control. This specifies whether the coordinates on the terminal's horizontal axis increase or decrease as you move from left to right across the screen.
	0 = Coordinates increase from left to right.
	1 = Coordinates decrease from left to right.
12	Vertical axis orientation control. This specifies whether the coordinates on the terminal's vertical axis increase or decrease as you move from top to bottom across the screen.
	0 = Coordinates increase from top to bottom.
	1 = Coordinates decrease from top to bottom.
13-15	Reserved bits. For future compatibility, set to 0.

NOTE

If bits 4-5 contain 2 or 3, and bits 6-8 also contain 2 or 3, then they must both contain the same value. That is, they must both reflect the same parity convention (even or odd).

IN$RATE

WORD indicating the input baud rate. The word is encoded as follows:

0 =	Invalid.
1 =	Perform an automatic baud rate search.
Other =	Actual input baud rate, such as 9600.

OUT$RATE

WORD indicating the output baud rate. The word is encoded as follows:

0 =	Use the input baud rate for output.
Other =	Actual output baud rate, such as 9600.

	Most applications require the input and output baud rates to be equal. In such cases, use IN$RATE to set the baud rate and specify a zero for OUT$RATE.
SCROLL$NUMBER	WORD specifying the number of lines that are to be sent to the terminal each time the operator enters the appropriate control character (Control-W is the default).
BUFFERED$DEVICE$-DATA	BYTES that contain additional information that applies to drivers of buffered devices (intelligent communications processors that maintain their own internal memory buffers). Refer to the "Additional Information for Buffered Devices" section to see how to access these bytes.

PROCEDURES THAT TERMINAL DRIVERS MUST SUPPLY

The routines that make up the Basic I/O System's Terminal Support Code constitute the bulk of the terminal device driver. These routines, in turn, make calls to device-dependent routines that you must supply. The following paragraphs describe the routines briefly. Sections that follow describe the routines in more detail.

A terminal initialization procedure. This procedure must perform any initialization functions necessary to get the terminal controller ready to process I/O requests. TSINITIO calls this procedure.

A terminal finish procedure. This procedure must perform any final processing so that the terminal controller can be detached. TSFINISHIO calls this procedure.

A terminal setup procedure. This procedure sets up the terminal in the proper mode (baud rate, parity, etc.). TSQUEUEIO and the Terminal Support Code's interrupt task call this procedure.

A terminal answer procedure. This procedure sets the Data Terminal Ready (DTR) line for modem support. TSQUEUEIO and the Terminal Support Code's interrupt task call this procedure.

A terminal hangup procedure. This procedure clears the Data Terminal Ready (DTR) line for modem support. TSQUEUEIO and the Terminal Support Code's interrupt task call this procedure.

A terminal check procedure. This procedure determines which terminal sent an interrupt signal and what type of interrupt it is. The Terminal Support Code's interrupt handler calls this procedure.

A terminal output procedure. This procedure displays a character at a terminal. TSQUEUEIO and the Terminal Support Code's interrupt task call this procedure.

A set output waiting procedure. This procedure signals the Terminal Support Code that a terminal is ready to perform character transmission and interrupt handling.

When the Terminal Support Code calls these procedures, it passes, as a parameter, a pointer to the TSC Data Area described in the previous section. If the called procedure is to perform duties on behalf of all of the terminals connected to the controller, the Terminal Support Code passes a pointer to the beginning of the TSC Data Area (the device portion). On the other hand, if the procedure is to perform duties for just a particular terminal, the Terminal Support Code passes a pointer to the unit portion of the TSC Data Area that corresponds to the terminal.

Because the TSC Data Area always starts on a paragraph boundary, a procedure that receives a pointer to a unit portion of the data area can construct a pointer to the beginning of the TSC Data Area. It does this by calling the PL/M-86 builtin procedure BUILD$PTR using the base part of the pointer it received and an offset of 0. Also, if a procedure, such as term$check, receives a pointer to the beginning of the TSC data area, it can calculate where any unit portion of the data area starts by using the following formula:

```
unit$data$p = base(of TSC data area):[30H + (unit number * 400H)]
```

TERMINAL INITIALIZATION PROCEDURE

This procedure must initialize the controller. The nature of this initialization is device-dependent. When finished, the terminal initialization procedure must fill in the STATUS field of the TSC Data Area, as follows:

- If initialization is successful, it must set STATUS to E$OK (0).
- If initialization is not successful, it should normally set STATUS equal to E$IO (2BH). However, it can set the STATUS field to any other value, in which case the Basic I/O System returns that value to the task that is attempting to attach the device. (The Human Interface ATTACHDEVICE command expects the procedure to return the E$IO status if initialization is unsuccessful.)

The syntax of a call to the user-written terminal initialization procedure is as follows:

```
CALL term$init(tsc$data$ptr);
```

where:

term$init	Name of the terminal initialization procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
tsc$data$ptr	POINTER to the beginning of the TSC Data Area.

TERMINAL FINISH PROCEDURE

The Terminal Support Code calls this procedure when a user detaches the last terminal unit on the terminal controller. The terminal finish procedure can simply do a RETURN, it can clean up data structures for the driver, or it can clear the controller. The syntax of a call to the user-written terminal finish procedure is as follows:

```
    CALL term$finish(tsc$data$ptr);
```

where:

term$finish	Name of the terminal finish procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
tsc$data$ptr	POINTER to the beginning of the TSC Data Area.

TERMINAL SETUP PROCEDURE

This procedure "sets up" one terminal according to the TERMINAL$FLAGS, IN$RATE, OUT$RATE, SCROLL$NUMBER, and BUFFERED$DEVICE$DATA fields in the corresponding UNIT$DATA portion of the TSC Data Area. In particular, if IN$RATE is 1, then the term$setup procedure must start a baud rate search. (The terminal check procedure usually finishes the search and then fills in IN$RATE with the actual baud rate.) If OUT$RATE is 0, the terminal setup procedure assumes the output baud rate is the same value as the input baud rate.

If your terminal controller is a buffered device (an intelligent device that manages its own internal data buffers), the terminal setup procedure must also set one of the reserved fields of the UNIT$DATA structure. Refer to the "Buffered Devices" section in this chapter for more information.

If your terminal driver supports a modem, the terminal setup procedure might have to perform additional services. Refer to the "Terminal Hangup" section for more information.

The terminal setup procedure must call the set output waiting procedure. Refer to a later section in this chapter for more information on the set output waiting procedure. The syntax of a call to the user-written terminal setup procedure is as follows:

```
    CALL term$setup(unit$data$n$ptr);
```

where:

term$setup	Name of the terminal setup procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
unit$data$n$ptr	POINTER to the terminal's UNIT$DATA structure in the TSC Data Area.

TERMINAL ANSWER PROCEDURE

This procedure activates the Data Terminal Ready line for a particular terminal. The Terminal Support Code calls the terminal answer procedure only when both of the following conditions are true:

- Bit 3 of TERMINAL$FLAGS in the terminal's UNIT$DATA structure (the modem indicator) is set to 1.
- The Terminal Support Code has received a Ring Indicate signal (the phone is ringing) or an answer request (via an OSC modem answer sequence) for the terminal. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information about OSC sequences.

The syntax of a call to the user-written terminal answer procedure is as follows:

```
    CALL term$answer(unit$data$n$p);
```

where:

term$answer	Name of the terminal answer procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
unit$data$n$p	POINTER to the terminal's UNIT$DATA structure in the TSC Data Area.

TERMINAL HANGUP PROCEDURE

This procedure clears the Data Terminal Ready line for a particular terminal. The Terminal Support Code calls the terminal hangup procedure only when both of the following are true:

- Bit 3 of TERMINAL$FLAGS in the terminal's UNIT$DATA structure (the modem indicator) is set to 1.
- The Terminal Support Code has received a Carrier Loss signal (the phone is hung up) or a hangup request (via an OSC modem hangup sequence) for the terminal. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information about OSC sequences.

The syntax of a call to the user-written terminal hangup procedure is as follows:

```
CALL term$hangup(unit$data$n$p);
```

where:

term$hangup	Name of the terminal hangup procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
unit$data$n$p	POINTER to the terminal's UNIT$DATA structure in the Terminal Support Code data Area.

NOTE

Some modem devices recognize only carrier detect as an indication that someone is calling and loss of carrier detect as an indication of hangup. However, most of these devices require the Data Terminal Ready line to be active before they can recognize carrier detect. For these devices, the terminal setup procedure must activate the Data Terminal Ready line. Likewise, the terminal hangup procedure must clear the Data Terminal Ready line and then reactivate it.

TERMINAL CHECK PROCEDURE

The Terminal Support Code calls this procedure whenever an interrupt occurs, which usually signals that a key on that terminal's keyboard has been pressed. When called, the terminal check procedure should determine the kind of interrupt and the interrupting unit, as follows:

1. Check all terminals on the device for an input character.

2. If no input character is available, check for a transmitter ready to send another character.

3. If no transmit character is available, check for a change in status (such as a ring or carrier interrupt).

When the terminal check procedure finds the first valid interrupt, it should quit scanning other units. Then it should place the unit number of the interrupting unit in the INTERRUPTING$UNIT field of the TSC Data Area and information about the type of interrupt in the INTERRUPT$TYPE field. The Terminal Support Code interprets values in the INTERRUPT$TYPE field as follows:

0	no interrupt
1	input interrupt
2	output interrrupt
3	ring interrupt
4	carrier interrupt
5	delay interrupt

Also, if the terminal check procedure detects another interrupt while it is returning information about the first interrupt, it should add the following value:

8	more interrupts

to the value it places in the INTERRUPT$TYPE field. Adding this value signals the Terminal Support Code to call the terminal check procedure again after it processes the current interrupt.

Unless the controller hardware guarantees that an additional interrupt will be set after one of multiple pending interrupts is serviced, the terminal check procedure should always signal that more interrupts are available unless it cannot detect interrupts at all. That is, it should always return one of the following values in the INTERRUPT$TYPE field:

0H	no interrupt
9H	input interrupt plus more
0AH	output interrupt plus more
0BH	ring interrupt plus more
0CH	carrier interrupt plus more
0DH	delay interrupt plus more

By returning these values, the terminal check procedure ensures that the Terminal Support Code calls it again. Otherwise, the driver could lose characters. If, in fact, there are no more interrupts to service, the terminal check procedure can return a zero value (no interrupt) the last time it is called.

If your terminal driver supports a baud rate search to determine the baud rate of an individual terminal, the terminal check procedure must ascertain the terminal's baud rate, as follows:

1. The first time the terminal check procedure encounters an input interrupt for a particular terminal, it should examine the IN$RATE field of that terminal's UNIT$DATA structure to determine the baud rate.

2. If the IN$RATE field is set to 1 (perform automatic baud rate search), the terminal check procedure should examine the input character to determine if it is an uppercase "U". (It can usually check for 19200, 9600, and 4800 baud in one attempt.)

3. If the terminal check procedure determines the baud rate, it should set the IN$RATE field of the UNIT$DATA structure to reflect the actual input baud rate.

4. If the terminal check procedure cannot determine the baud rate, it should increment the IN$RATE field in the UNIT$DATA structure. When the next input interrupt occurs, the terminal check procedure can try again to determine the baud rate. Refer to the example terminal driver in Appendix B to see how to implement a baud rate scan.

5. Place a value of ODH in the INTERRUPT$TYPE field (delay interrupt plus more). The ODH value tells the Terminal Support Code that a baud rate scan is in progress. The Terminal Support Code then waits a few clock cycles and calls the terminal setup procedure to "set up" the terminal for the new baud rate.

If the terminal check procedure encounters an input interrupt, it must also return the input character to the procedure that called it, adjusting the parity bit according to bits 4 and 5 of the TERMINAL$FLAGS field in the interrupting unit's UNIT$DATA structure. If the interrupt is not an input interrupt, the terminal check procedure can return any value.

The syntax of the call to the user-written terminal check procedure is as follows:

```
    input$char = term$check(tsc$data$ptr)
```

where:

input$char	BYTE in which the terminal check procedure returns the input character, if the interrupt was an input interrupt. If the interrupt was not an input interrupt, this parameter can have any value.

term$check	Name of the terminal check procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
tsc$data$ptr	POINTER to the start of the Terminal Support Code Data Area.

TERMINAL OUTPUT PROCEDURE

The Terminal Support Code calls this procedure to display a character at a terminal. The Terminal Support Code passes it the character and a pointer to the terminal's UNIT$DATA structure. If bits 6 through 8 of the TERMINAL$FLAGS field of the UNIT$DATA structure so indicate, the terminal output procedure should adjust the character's parity bit and then output the character to the terminal.

The syntax of the call to the user-written terminal output procedure is as follows:

```
    CALL term$out(unit$data$n$p, output$character);
```

where:

term$out	Name of the terminal output procedure. You can use any name for this procedure, as long as it doesn't conflict with other procedure names and you include the name in the Device Information Table.
unit$data$n$p	POINTER to the terminal's UNIT$DATA structure in the TSC Data Area.
output$character	BYTE containing a character that the terminal output procedure should send to the terminal.

SET OUTPUT WAITING PROCEDURE

This procedure notifys the Terminal Support Code that the particular terminal is ready to perform data transmission.

The syntax of a call to the set output waiting procedure is as follows:

```
    CALL xts$set$output$waiting (unit$data$n$p);
```

where:

xts$set$output $waiting	Name of the Terminal Support Code provided procedure. The terminal setup procedure that you write must declare xts$set$output$waiting as an external procedure with one pointer parameter.
unit$data$n$ptr	POINTER to the terminal's UNIT$DATA structure in the TSC Data Area, This si the same pointer passed to the terminal setup procedure by the Terminal Support Code.

ADDITIONAL INFORMATION FOR BUFFERED DEVICES

If you are writing a driver for a buffered communications device (an intelligent communications processor like the iSBC 544 board that manages its own buffers of data separately from the ones managed by the Terminal Support Code), your driver routines must make use of the BUFFERED$DEVICE$DATA fields of the UNIT$DATA structure. In so doing, they should impose the following structure on those 11 bytes:

```
DECLARE  BUFFERED$DEVICE$DATA  STRUCTURE(
    BUFFERED$DEVICE       BYTE,
    FLOW$CONTROL          WORD,
    HIGH$WATER$MARK       WORD,
    LOW$WATER$MARK        WORD,
    FC$ON$CHAR            WORD,
    FC$OFF$CHAR           WORD);
```

where:

BUFFERED$DEVICE — When true, a BYTE that specifies whether the unit requires handling as a buffered device.

FLOW$CONTROL — WORD specifying whether the communications board sends flow control characters (selected by the FC$ON$CHAR and FC$OFF$CHAR fields, but usually XON and XOFF) to turn input on and off. The low-order bit (bit 0) controls this option, as follows:

0 Disable flow control.

1 Enable flow control.

When flow control is enabled, the communication board can control the amount of data sent to it to prevent buffer overflow.

HIGH$WATER$MARK — When the communication board's input buffer fills to contain the number of bytes specified in this WORD, the board sends the flow control character to stop input.

LOW$WATER$MARK	When the number of bytes in the communication board's input buffer drops to the number specified in this WORD, the board sends the flow control character to start input.
FC$ON$CHAR	WORD specifying an ASCII character that the communication board sends to the connecting device when the number of bytes in its buffer drops to the low-water mark. Normally this character tells the connecting device to resume sending data.
FC$OFF$CHAR	A WORD specifying an ASCII character that the communication board sends to the connecting device when the number of characters in its buffer rises to the high-water mark. Normally this character tells the connecting device to stop sending data.

When a user attaches a unit on any terminal device, the Terminal Support Code calls the terminal setup procedure. If the device is a buffered device, the terminal setup procedure must set the BUFFERED$DEVICE field to TRUE (OFFH). It should also fill in the other fields of the BUFFERED$DEVICE$DATA structure. In addition, it should enable the communication device's on-board receiver interrupt (the one for the unit being attached) so that it can accept data from the connected terminal.

When a user detaches a unit on a buffered device, the Terminal Support Code sets the BUFFERED$DEVICE field to FALSE (OH) and again calls the terminal setup procedure. The terminal setup procedure should disable the communication device's on-board receiver interrupt (the one for the unit being detached) to prevent extraneous characters from being received.

To distinguish between an "attach device" and a "detach device", the terminal setup procedure should establish its own internal flags (one for each unit) in addition to the BUFFERED$DEVICE fields. It can use these flags as follows:

1. Initially, the terminal initialization procedure sets the flag of each unit to FALSE to indicate that no devices are attached.

2. When the Terminal Support Code calls the terminal setup procedure to attach a unit, both the BUFFERED$DEVICE field and the internal flag are FALSE. The terminal setup procedure recognizes from this combination that the operation is an "attach device."

3. The terminal setup procedure performs the "attach device" operations and sets the internal flag and the BUFFERED$DEVICE flag to TRUE to indicate that the device is attached.

4. When the unit is detached, the Terminal Support Code sets the BUFFERED$DEVICE flag to FALSE and calls the terminal setup procedure. In this situation, the BUFFERED$DEVICE field is FALSE, but the internal flag is TRUE. The terminal setup procedure recognizes from this combination that the operation is a "detach device."

PROCEDURES' USE OF DATA STRUCTURES

Table 7-1 helps you sort out the responsibilities of the various procedures in a terminal device driver. In the table, the following codes refer to those procedures:

(1) terminal initialization
(2) terminal finish
(3) terminal setup
(4) terminal answer
(5) terminal hangup
(6) terminal check
(7) terminal output

Also, "System" and "ICU" are used in Table 7-1 to indicate the iRMX 86 software and the iRMX 86 Interactive Configuration Utility, respectively. In addition, "Term$flags" is an abbreviation of "Terminal$flags," and numbers following immediately after "Term$flags" are bit numbers in that word.

Table 7-1. Uses of Fields in Terminal Driver Data Structures

	Filled in/Changed by	Can or Will be Used by
TSC$DATA		
IOS$DATA$SEGMENT	System	(1)-(7)
STATUS	(1)	System
INTERRUPT$TYPE	(6)	System
INTERRUPTING$UNIT	(6)	System
DEV$INFO$PTR	System	(1)-(7)
USER$DATA$PTR	System	(1)-(7)
UNIT$DATA		
UNIT$INFO$PTR	System	System
TERM$FLAGS (0-2)	System	System
TERM$FLAGS (3)	System	(3)
TERM$FLAGS (4-5)	System	(3),(6)
TERM$FLAGS (6-8)	System	(3),(6),(7)
IN$RATE	System,(3),(6)	(3)
OUT$RATE	System	(3)
SCROLL$NUMBER	System	System
BUFFERED$DEVICE$DATA	(3)	System, (3)
TERMINAL$DEVICE$INFORMATION		
NUM$UNITS	ICU	System
DRIVER$DATA$SIZE	ICU	System
STACK$SIZE	ICU	System
TERM$INIT	ICU	System
TERM$FINISH	ICU	System
TERM$SETUP	ICU	System
TERM$OUT	ICU	System
TERM$ANSWER	ICU	System
TERM$HANGUP	ICU	System
TERM$CHECK	ICU	System
INTERRUPTS		
INTERRUPT$LEVEL	ICU	System
TERM$CHECK	ICU	System
DRIVER$INFO	ICU	(1)-(7)

***

CHAPTER 8 BINDING A DEVICE DRIVER TO THE I/O SYSTEM

You can write the modules for your device driver in either PL/M-86 or the ASM86 Macro Assembly Language. However, you must adhere to the following guidelines:

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

- If you use assembly language, your routines must follow the conditions and conventions used by the PL/M-86 COMPACT size control. In particular, your routines must function in the same manner as reentrant PL/M-86 procedures with the ROM and COMPACT controls set. The ASM86 MACRO ASSEMBLER OPERATING INSTRUCTIONS manual describes these conditions and conventions.

USING THE iRMX™ 86 INTERACTIVE CONFIGURATION UTILITY

To use the iRMX 86 Interactive Configuration Utility to configure a driver that you have written for your application system, you must perform the following steps:

1. For each device driver that you have written, assemble or compile the code for the driver.

2. Put all the resulting object modules in a single library, such as DRIVER.LIB.

3. Ascertain the device numbers and device-unit numbers to use in the DUIBs for your devices.

   a. Use the ICU to configure a system containing all the Intel-supplied drivers you require.

   b. Use the G option to generate that system.

   c. Use a text editor to examine the file IDEVCF.A86. Among other things, this file contains DUIBs for all the device-units you defined in your configuration.

   d. Look for the DEFINE_DUIB structures in the file. Chapter 2 lists the format of these structures. Note the device number (eighth field) and the device-unit number (tenth field) of the last DUIB defined in the file.

   Figure 8-1 lists part of an IDEVCF.86 file which contains this information (the file you examine might look different, depending on how you configure your system). The arrows in the figure point to the relevant fields.

e. Use the next available device numbers and device-unit numbers in your DUIBs.

```
                 .
                 .
                 .
            DEFINEDUIB <
            & 'lp',
            & 00001H,
            & 0F2H,
            & 00,
            & 00,
            & 00,
            & 00,
    ---->   & 00004H,
            & 00,
    ---->   & 0000BH,
            & INITIO,
            & FINISHIO,
            & QUEUEIO,
            & CANCELIO,
            & DINFO04,
            & 00,
            & 0FFFFH,
            & 00000H,
            & 130,
            & FALSE,
            & 00000H,
            & 0
            &>
            NUMDUIB EQU (THIS BYTE - DUIBTABLE) / SIZE DEFINEDUIB
            BIOSCODE ENDS
            %DEVICETABLES(NUMDUIB,0000CH,005H,003E8H)
            CODE SEGMENT
            ASSUME CS:CGROUP
                 .
                 .
                 .
```

Figure 8-1. Example IDEVCF.A86 File

4. Create the following:

   a. A file containing the DUIBs for all the device-units you are adding. Use the DEFINE_DUIB structures shown in Chapter 2. Place all the structures in the same file. Later, the ICU includes this file in the assembly of the IDEVCF.A86 file.

   b. A file containing all the device information tables you are adding. Use the RADEV_DEV_INFO structures shown in Chapter 2 for any random access drivers you add. Later, the ICU includes this file in the assembly of the IDEVCF.A86 file.

   c. If applicable, any unit information table(s). Use the RADEV_UNIT_INFO structures shown in Chapter 2 for any random access drivers you add. Add these tables to the file created in step b.

   d. External declarations for any procedures that you write. The names of these procedures appear in either the DUIB or the Device Information Table associated with this device driver. Add these declarations to the file created in step b.

5. Use the ICU to configure your final system. When doing so:

   a. Answer "yes" when asked if you have any device drivers not supported by the ICU (this means drivers that you have written).

   b. As input to the "User Devices" screen, enter the pathname of your device driver library. This refers to the library built in step 2; for example, :Fl:DRIVER.LIB.

   c. Also, enter the information the ICU needs to include your configuration data in the assembly of IDEVCF.A86. The information needed includes the following:

      - DUIB source code pathname (the file created in step 4a).

      - Device and Unit source code pathname (the file created in steps 4b through 4d).

      - Number of user defined devices.

      - Number of user defined device-units.

The ICU does the rest.

Figure 8-2 contains an example of the "User Devices" screen. The underlined text represents user input to the ICU. In this example, the file :Fl:DRIVER.LIB contains the object code for the driver, :Fl:DUIB.SRC contains the source code for the DUIBs, and :Fl:DEVINF.SRC contains the source code for the Device and Unit Information Tables along with the necessary external procedure declarations.

The code in the DRIVER.LIB file supports one device with two units. Refer to the iRMX 86 CONFIGURATION GUIDE for instructions on how to use the ICU.

```
User Devices
(OPN) Object Code Path Name [1-45 characters]
                                NONE
(DPN) Duib Source Code Path Name [1-45 characters]

(DUP) Device and Unit Source Code Path Name [1-45 characters]

(ND)  Number of User Defined Devices [0-0FFH]              0001H
(NDU) Number of User Defined Device-Units [0-0FFH]         0001H

Enter Changes [Abbreviations ?/= new_value] : OPN = :F1:DRIVER.LIB
: DPN = :F1:DUIB.SRC
: DUP = :F1:DEVINF.SRC
: ND = 1
: NDU = 2
```

Figure 8-2. Example User Devices Screen

USING THE iRMX™ 88 INTERACTIVE CONFIGURATION UTILITY

To use the iRMX 88 Interactive Configuration Utility to configure a driver that you have written for your application system, you must perform the following steps in the following order:

1. For each driver, assemble or compile the code.

2. When using the ICU:

   a. Answer "208", "215", "common", "random", or "custom" when asked for device type.

   b. When prompted, enter the information for the DUIBs, the device information tables, and, if applicable, the unit information table.

   c. When prompted for linking information, enter the names of the appropriate modules.

The ICU does the rest.

***

APPENDIX A RANDOM ACCESS DRIVER SUPPORT ROUTINES

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

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

NOTE

For iRMX 88 systems, these names are prefixed by "RAD$".

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 to support a random access or 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 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 to make writing the device-dependent portion easier (the random access driver support routines follow essentially the same pattern). Or you can use it as a guideline for writing custom device drivers.

INIT$IO PROCEDURE

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

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

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

- A pointer to the DUIB of the device-unit to initialize
- In the iRMX 86 environment, a pointer to the location where INIT$IO must return a token for a data segment (data storage area) that it creates
- A pointer to the location where INIT$IO must return the condition code

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

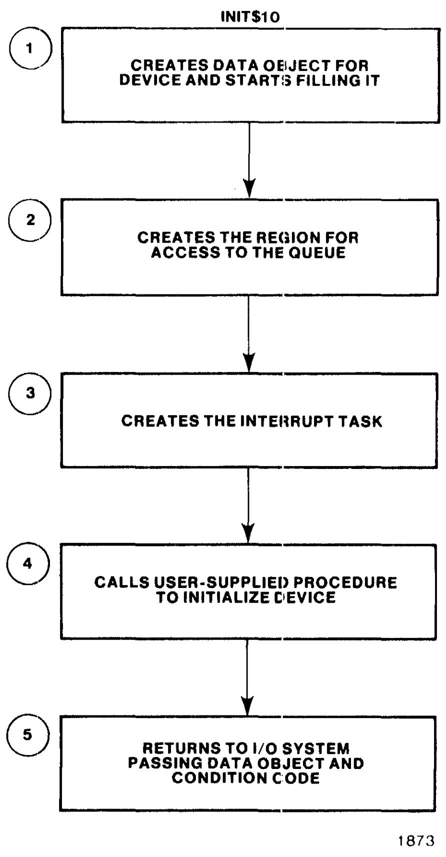
INIT$IO
1
CREATES DATA OBJECT FOR DEVICE AND STARTS FILLING IT
2
CREATES THE REGION FOR ACCESS TO THE QUEUE
3
CREATES THE INTERRUPT TASK
4
CALLS USER-SUPPLIED PROCEDURE TO INITIALIZE DEVICE
5
RETURNS TO I/O SYSTEM PASSING DATA OBJECT AND CONDITION CODE
1873

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

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

   - The value of the DS (data segment) register.
   - A token (identifier) for a region (exchange) --- for mutual exclusion.
   - An array which will contain the addresses of the DUIBs for the device-units attached to this device. INIT$IO places the address of the DUIB for the first attaching device unit to this array.
   - A token (identifier) for the interrupt task.
   - Other values indicating that the queue is empty and the driver is not busy.

   It also reserves space in the data storage area for device data.

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

3. It creates an interrupt task to handle interrupts generated by this device. INIT$IO passes to the interrupt task a token for the data storage area. This area is where the interrupt task will get information about the device. Also, INIT$IO places a token for the interrupt task in the data storage area.

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

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

FINISH$IO PROCEDURE

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

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

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

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

The following paragraphs show the general steps that the FINISH$IO procedure goes through 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 specified in the DUIB. Refer to the Chapter 4 for information about device information tables.

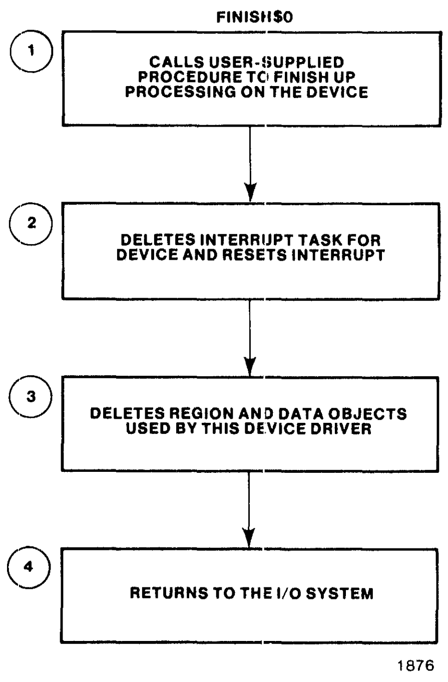
FINISH$O
1
CALLS USER-SUPPLIED PROCEDURE TO FINISH UP PROCESSING ON THE DEVICE
2
DELETES INTERRUPT TASK FOR DEVICE AND RESETS INTERRUPT
3
DELETES REGION AND DATA OBJECTS USED BY THIS DEVICE DRIVER
4
RETURNS TO THE I/O SYSTEM
1876

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

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

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

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

QUEUE$IO PROCEDURE

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

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

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

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

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

2. It receives control of the region and thus access to the queue. 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 to start the processing of the request at the head of the queue. This results in a call to a user-written device start procedure which actually sends the data to the device itself. This start procedure is described in Chapter 5. If the device is already busy processing some other request, this step does not start the data transfer.

5. It surrenders control of the region, thus allowing other routines to have access to the queue.

CANCEL$IO PROCEDURE

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

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

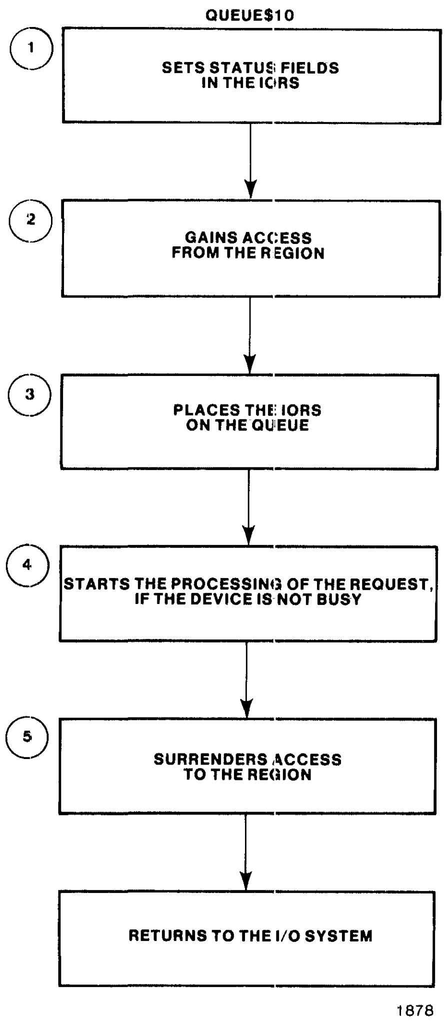
QUEUE$IO
1
SETS STATUS FIELDS IN THE IORS
2
GAINS ACCESS FROM THE REGION
3
PLACES THE IORS ON THE QUEUE
4
STARTS THE PROCESSING OF THE REQUEST, IF THE DEVICE IS NOT BUSY
5
SURRENDERS ACCESS TO THE REGION
RETURNS TO THE I/O SYSTEM
1878

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

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

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

- An ID value that identifies requests to be cancelled
- A pointer to the DUIB
- A token (identifier) for the device data storage area

The following paragraphs show the general steps that the CANCEL$IO procedure goes through 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 by gaining control of the region. This allows it to remove requests from the queue without concern that other tasks might also be processing the IORS at the same time.

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

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

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

5. It surrenders control of the region, thus allowing other procedures to gain access to the queue.

NOTE

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

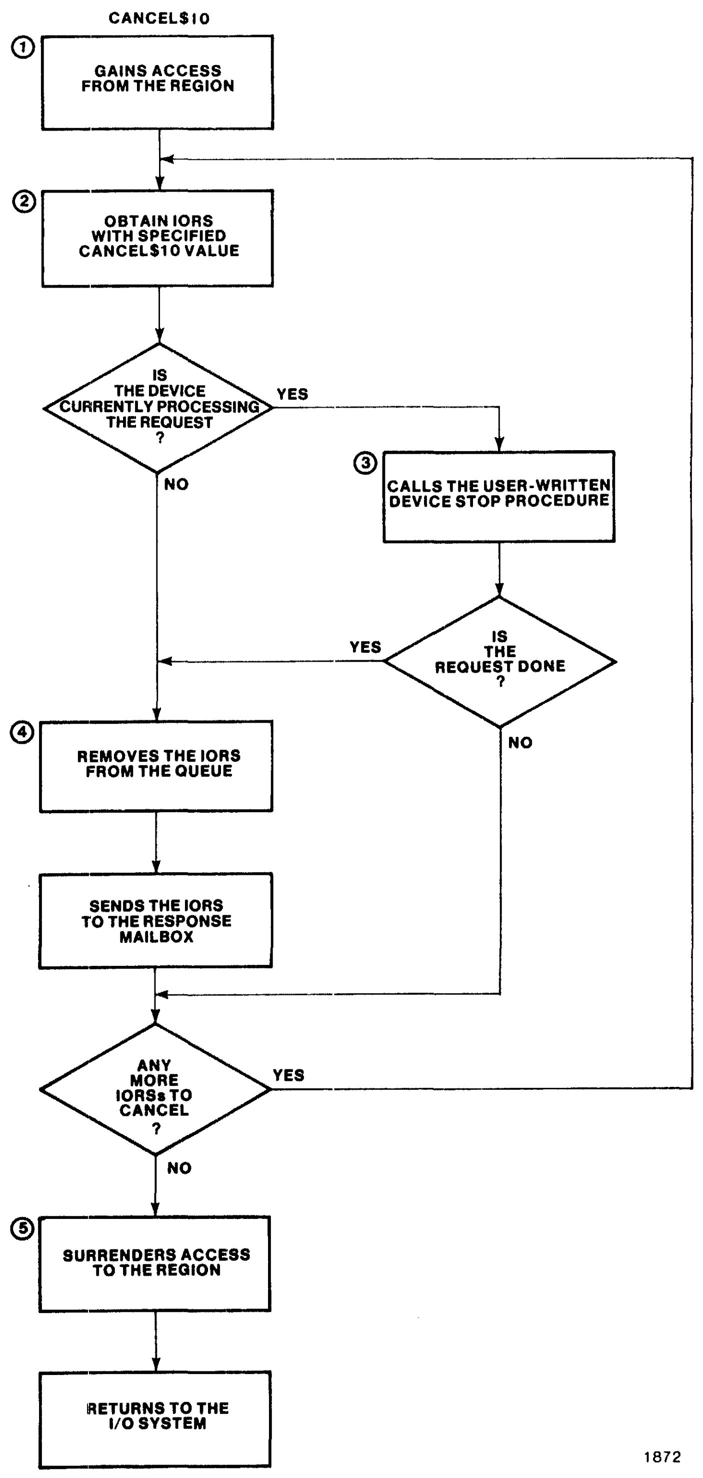
CANCEL$10
1
GAINS ACCESS FROM THE REGION
2
OBTAIN IORS WITH SPECIFIED CANCEL$10 VALUE
IS THE DEVICE CURRENTLY PROCESSING THE REQUEST ?
YES
NO
3
CALLS THE USER-WRITTEN DEVICE STOP PROCEDURE
IS THE REQUEST DONE ?
YES
NO
4
REMOVES THE IORS FROM THE QUEUE
SENDS THE IORS TO THE RESPONSE MAILBOX
ANY MORE IORSs TO CANCEL ?
YES
NO
5
SURRENDERS ACCESS TO THE REGION
RETURNS TO THE I/O SYSTEM
1872

Figure A-4. Random Access 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 random access device driver goes through 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 processor's DS register to obtain a token (identifier) for the device data storage area. This is possible because of the following two reasons:

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

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

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

2. For iRMX 86 systems, it makes an RQ$SET$INTERRUPT system call to indicate that it is an interrupt task associated with the interrupt handler supplied with the random access device driver. It also indicates the interrupt level to which it will respond.

   For iRMX 88 systems, it makes an RQ$ELVL system call to enable the nucleus-provided default interrupt handler.

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.

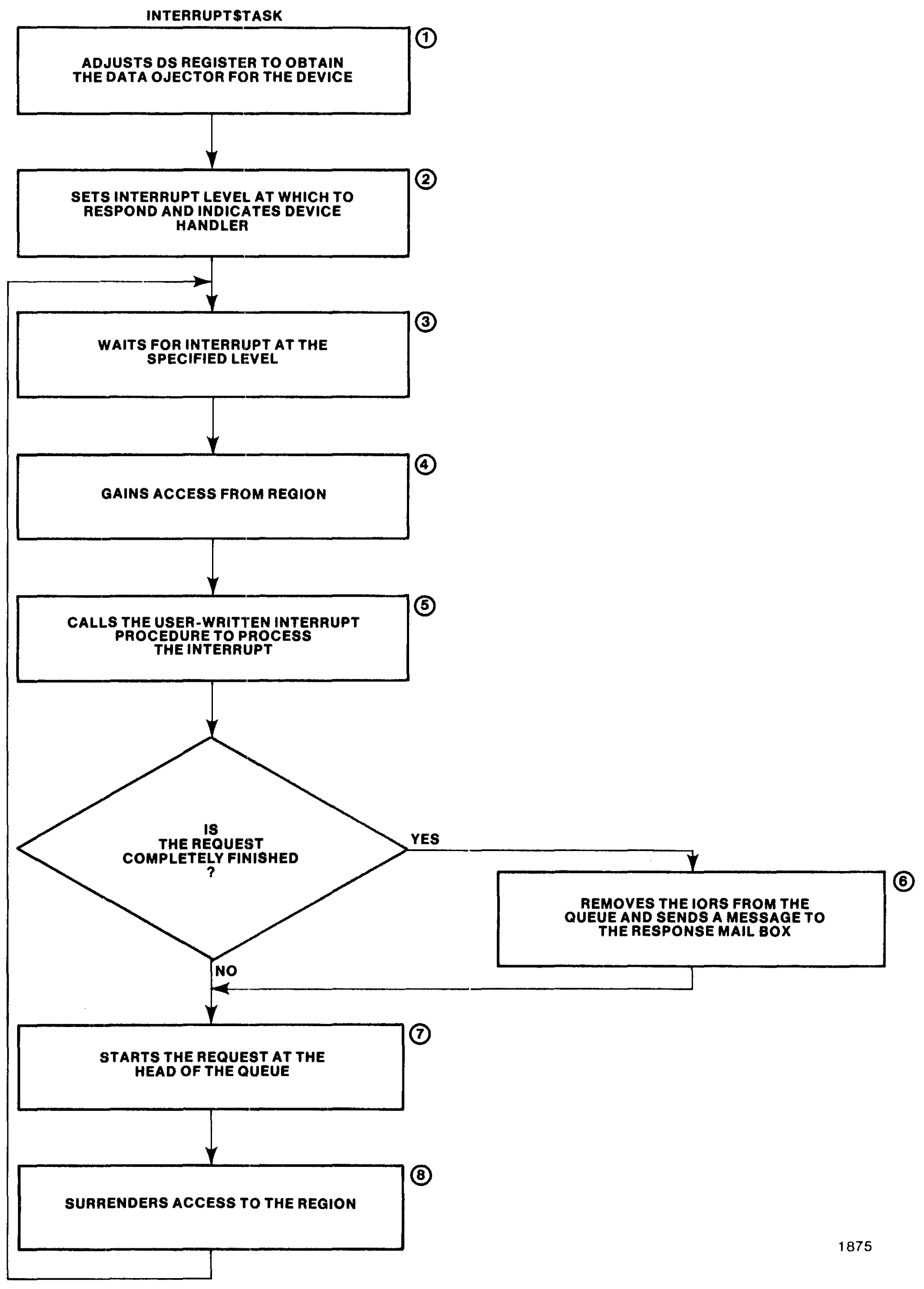
INTERRUPT$TASK
ADJUSTS DS REGISTER TO OBTAIN THE DATA OJECTOR FOR THE DEVICE
1
SETS INTERRUPT LEVEL AT WHICH TO RESPOND AND INDICATES DEVICE HANDLER
2
WAITS FOR INTERRUPT AT THE SPECIFIED LEVEL
3
GAINS ACCESS FROM REGION
4
CALLS THE USER-WRITTEN INTERRUPT PROCEDURE TO PROCESS THE INTERRUPT
5
IS THE REQUEST COMPLETELY FINISHED ?
YES
REMOVES THE IORS FROM THE QUEUE AND SENDS A MESSAGE TO THE RESPONSE MAIL BOX
6
NO
STARTS THE REQUEST AT THE HEAD OF THE QUEUE
7
SURRENDERS ACCESS TO THE REGION
8
1875

Figure A-5. Random Access Device Driver Interrupt Task

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

7. If there are requests on the queue, the interrupt task initiates the processing of the next I/O request by calling the user-written device-start procedure.

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

***

APPENDIX B EXAMPLES OF DEVICE DRIVERS

This appendix contains four examples of device drivers. The first example is a common driver which drives a line printer. The second is a random access driver, which drives a iSBC 206 disk controller. The third example is an 8274 terminal driver. (The contents of the INCLUDE files that these drivers use are listed in the last section of this appendix.)

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

Table B-1 lists the device driver example file names and the pages on which they appear.

Table B-1. Device Driver Examples

File	Description	Page
iprntr.p86	Driver for a line printer	B-2
i206ds.p86	Driver for an iSBC 206 disk controller	B-6
x8274.p86	8274 terminal driver	B-20
	INCLUDE files for above device drivers	B-39

```
PL/M-86 COMPILER    xprntr.p86


iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE XPRNTR
OBJECT MODULE PLACED IN :F1:XPRNTR.OBJ
COMPILER INVOKED BY:  :LANG:plm86 :F1:XPRNTR.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE



                $title ('xprntr.p86')
                /*
                 *  xprntr.p86
                 *
                 *     This module implements centronix-type interface line printer
                 *     driver.  It is written as a 'common' device driver.  It is
                 *     assumed that the reader is familiar with the 8255 chip.
                 *
                 *
                 *  LANGUAGE DEPENDENCIES:
                 *     COMPACT ROM OPTIMIZE(3)
                 */

                /*
                 *             INTEL CORPORATION PROPRIETARY INFORMATION
                 *
                 *          This software is supplied under the terms of a
                 *          license agreement or nondisclosure agreement with
                 *          Intel Corporation and may not be copied or disclosed
                 *          except in accordance with the terms of that agreement.
                 *
                 */

  1             xprntr: DO;
                $include(:f1:xcomon.lit)
          =     $save nolist
                $include(:f1:xparam.lit)
          =     $save nolist
                $include(:f1:xnutyp.lit)
          =     $save nolist
                $include(:f1:xiors.lit)
          =     $save nolist
                $include(:f1:xduib.lit)
          =     $save nolist
                $include(:f1:xprntr.lit)
          =     $save nolist
                $include(:f1:x8255.lit)
          =     $save nolist
                $include(:f1:xprerr.lit)
          =     $save nolist
                $include(:f0:nsleep.ext)
          =     $SAVE NOLIST
                /*
                 *  literal declaration
                 */
 23   1         DECLARE
                   TAB$CHAR LITERALLY '09H',
                   SPACE    LITERALLY '20H';


                $subtitle('printer$start$interrupt')

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

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

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

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

   DO CASE (iors.funct);

      /* read */
      DO;
         iors.status = E$IDDR;
         iors.done = TRUE;
      END;

      /* write */
      DO;
         /* get the buffer pointer */
         buffer$p = iors.buff$p;




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

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

            /*
             *  test for printer ready and not paper out. if not ready
             *  or paper out then wait forever.
             */
            DO WHILE (((INPUT(dinfo.C$port) AND PRINTER$READY) = 0) OR
                      ((INPUT(dinfo.C$port) AND PAPER$OUT) <> 0));
               /*  sleep for 100 nucleus clock intervals */
               CALL rq$sleep(100, @iors.status);
            END;
      /*
       *  convert TAB character to a SPACE character if the
       *  printer does not handle them
       */
      IF ((char(iors.actual) = TAB$CHAR) AND
                                 ((dinfo.tab$control) = FALSE))
          THEN char(iors.actual) = SPACE;
      /*
       * 1's complement the character and send it to the
       * printer.  Port-A is the data port
       */
      OUTPUT(dinfo.A$port) = NOT(char(iors.actual));
      /*
       * strobe the line printer
       * this is a way of telling the printer that there is
       * valid data on the bus
       */
      OUTPUT(dinfo.Control$port) = STROBE$ON;
      OUTPUT(dinfo.Control$port) = STROBE$OFF;
      /*
       *  increment the count of chars printed
       */
      iors.actual = iors.actual + 1;
      /*
       *  test whether printer acknowledgement bit is set
       */
      IF (INPUT(dinfo.C$port) AND CHAR$ACK AND CHAR$ACK$COMPLETE) = 0
         THEN
      DO;
         /*
          *  printer didn't acknowledge. Hopefully it has
          *  started printing. So enable the printer interrupt
          *  and return(printer will interrupt when it's done)
          */
         OUTPUT(dinfo.Control$port) = INT$ENABLE;
         RETURN;
      END;
   END;  /* end of DO WHILE statement */

   /*
    *  set iors.done to TRUE
    *  set iors.status to OK
    */
   iors.status = E$OK;
   iors.done = TRUE;
END;

/* seek */
DO;
   iors.status = E$IDDR;
   iors.done = TRUE;
END;

/* special */
DO;
   iors.status = E$IDDR;
   iors.done = TRUE;
END;

/* attach device */
DO;
   /* initialize the 8255 */
   OUTPUT(dinfo.Control$port) = MODE$WORD;
   iors.status = E$OK;
   iors.done = TRUE;
END;
```

```
      /* detach device */
      DO;
         iors.status = E$OK;
         iors.done = TRUE;
      END;

      /* open */
      DO;
         iors.status = E$OK;
         iors.done = TRUE;
      END;

      /* close */
      DO;
         iors.status = E$OK;
         iors.done = TRUE;
      END;

   END;  /* end of DO CASE statement */

END printer$start$interrupt;


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

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

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

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

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

END printer$stop;

END xprntr;
```

```
PL/M-86 COMPILER    x206ds.p86
                    Module Header


iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE X206DS
OBJECT MODULE PLACED IN :F1:X206DS.OBJ
COMPILER INVOKED BY:  :LANG:plm86 :F1:X206DS.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE


            $title('x206ds.p86')
            $subtitle('Module Header')

            /*
             *  x206ds.p86
             *
             *  iSBC 206 device
             *
             * LANGUAGE DEPENDENCIES: COMPACT ROM OPTIMIZE(3)
             */

   1        x206ds: DO;


            /*
             *             INTEL CORPORATION PROPRIETARY INFORMATION
             *
             *          This software is supplied under the terms of a
             *          license agreement or nondisclosure agreement with
             *          Intel Corporation and may not be copied or disclosed
             *          except in accordance with the terms of that agreement.
             *
             */

            $include(:f1:xcomon.lit)
       =    $save nolist
            $include(:f1:xnutyp.lit)
       =    $save nolist
            $include(:f1:xparam.lit)
       =    $save nolist
            $include(:f1:xiotyp.lit)
       =    $save nolist
            $include(:f1:xiors.lit)
       =    $save nolist
            $include(:f1:xduib.lit)
       =    $save nolist
            $include(:f1:xdrinf.lit)
       =    $save nolist
            $include(:f1:x206in.lit)
       =    $save nolist
            $include(:f1:x206dv.lit)
       =    $save nolist
            $include(:f1:xexcep.lit)
       =    $save nolist
            $include(:f1:xioexc.lit)
       =    $save nolist
            $include(:f1:xradsf.lit)
       =    $save nolist

            $include(:f1:x206dp.ext)
       =    $save nolist
            $include(:f1:x206dc.ext)
       =    $save nolist
            $include(:f1:x206fm.ext)
       =    $save nolist
            $include(:f1:xnotif.ext)
       =    $save nolist
```

```
          $subtitle('Local Data')

              /*
               *  need$reset array used to determine if device needs to be
               *  reset after an error. Indexed by status.
               */

 46   1       DECLARE
                  need$reset(24)      BYTE DATA(
                      FALSE,                      /* Successful completion */
                      TRUE,                       /* ID field miscompare */
                      FALSE,                      /* Data field CRC error */
                      FALSE,                      /* special for incorrect result$type */
                      TRUE,                       /* Seek error */
                      FALSE,
                      FALSE,
                      FALSE,
                      FALSE,                      /* Illegal Record Address */
                      FALSE,
                      FALSE,                      /* ID Field CRC error */
                      TRUE,                       /* Protocol error */
                      TRUE,                       /* Illegal Cylinder Address */
                      FALSE,
                      FALSE,                      /* Record not found */
                      FALSE,                      /* Data Mark Missing */
                      FALSE,                      /* Format Error */
                      FALSE,                      /* Write Protected */
                      FALSE,
                      TRUE,                       /* Write Error */
                      FALSE,
                      FALSE,
                      FALSE,
                      FALSE);                     /* Drive Not Ready */

              /*
               *  unit$status is used to set unit status field in iors.
               *  Indexed by status.
               */

 47   1       DECLARE
                  unit$status(24)     BYTE DATA(
                      IO$UNCLASS,                 /* Successful completion */
                      IO$SOFT,                    /* ID field miscompare */
                      IO$SOFT,                    /* Data field CRC error */
                      IO$HARD,                    /* special for incorrect result$type */
                      IO$SOFT,                    /* Seek error */
                      IO$UNCLASS,
                      IO$UNCLASS,
                      IO$UNCLASS,
                      IO$HARD,                    /* Illegal Record Address */
                      IO$UNCLASS,
                      IO$SOFT,                    /* ID Field CRC error */
                      IO$SOFT,                    /* Protocol error */
                      IO$HARD,                    /* Illegal Cylinder Address */
                      IO$UNCLASS,
                      IO$SOFT,                    /* Record not found */
                      IO$SOFT,                    /* Data Mark Missing */
                      IO$SOFT,                    /* Format Error */
                      IO$WRPROT,                  /* Write Protected */
                      IO$UNCLASS,
                      IO$SOFT,                    /* Write Error */
                      IO$UNCLASS,
                      IO$UNCLASS,
                      IO$UNCLASS,
                      IO$OPRINT);                 /* Drive Not Ready */

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

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

$subtitle('i206$start')

    /*
     * i206$start
     *      start procedure for the iSBC 206
     *
     * CALLING SEQUENCE:
     *      CALL i206$start(iors$p, duib$p, ddata$p);
     *
     * INTERFACE VARIABLES:
     *      iors$p      - I/O Request/Result segment pointer
     *      duib$p      - pointer to Device-Unit Information Block
     *      ddata$p     - device data segment pointer.
     *
     * CALLS:
     *      io$206
     *      format$206
     *      send$206$iopb
     *
     */

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

        dinfo$p = duib.device$info$p;
        base = dinfo.base;
        uinfo$p = duib.unit$info$p;

        IF (ddata.restore) THEN
            RETURN;

    do$case$funct:
        DO CASE iors.funct;

            /*
             * in the following calls the @ddata is literally
             * iopb$p (i.e., the pointer to the iopb).
             */

        case$read:
            DO;
                CALL io$206(base, iors$p, duib$p, @ddata);
            END case$read;

        case$write:
            DO;
                CALL io$206(base, iors$p, duib$p, @ddata);
            END case$write;

        case$seek:
            DO;
                CALL io$206(base, iors$p, duib$p, @ddata);
            END case$seek;
    case$spec$funct:
        DO;
            IF iors.sub$funct = FS$FORMAT$TRACK THEN
                CALL format$206(base, iors$p, duib$p, @ddata);
            ELSE
                DO;
                    iors.status = E$IDDR;
                    iors.actual = 0;
                    iors.done = TRUE;
                END;
        END case$spec$funct;

    case$attach$device:
        DO;
            dummy = (duib.dev$gran = 512);
            IF ((input(sub$system$port) OR 073H) <> 0FBH) OR
                (((input(disk$config$port) AND SHL(010H,SHR(duib.unit,2))) <> 0) <> dummy) THEN
                DO;
                    iors.status = E$IO;
                    iors.unit$status = IO$OPRINT;
                    iors.actual = 0;
                    iors.done = TRUE;
                    RETURN;
                END;
            ddata.inter = inter$on$mask;
            ddata.instr = restore$op;
            IF NOT send$206$iopb(base, @ddata) THEN
               /*
                * the board would not accept the iopb
                * so...
                */
                DO;
                    iors.status = E$IO;
                    iors.unit$status = IO$SOFT OR SHL(input(result$byte$port), 8);
                    iors.actual = 0;
                    iors.done = TRUE;
                END;
        END case$attach$device;

    case$detach$device:
        DO;
            iors.status = E$OK;
            iors.done = TRUE;
        END case$detach$device;

    case$open:

        DO;
            iors.status = E$OK;
            iors.done = TRUE;
        END case$open;

    case$close:
        DO;
            iors.status = E$OK;
            iors.done = TRUE;
        END case$close;

    END do$case$funct;

END i206$start;
```

```
$subtitle('i206$interrupt')

    /*
     * i206$interrupt
     *      interrupt procedure for the iSBC 206
     *
     * CALLING SEQUENCE:
     *      CALL i206$interrupt(iors$p, duib$p, ddata$p);
     *
     * INTERFACE VARIABLES:
     *      iors$p      - I/O Request/Result segment pointer
     *      duib$p      - pointer to Device-Unit Information Block
     *      ddata$p     - device data segment pointer.
     *
     * CALLS:
     *      i206$start
     *      send$206$iopb
     *      rq$send$message
     *
     */

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

        dinfo$p = duib.device$info$p;
        base = dinfo.base;
        spindle = shr(duib.unit,.2);                        /* 4 units/spindle */

        IF (input(result$type$port) AND 3) = 0 THEN
        done$int:
            DO;
                status = input(result$byte$port);

                IF ddata.restore THEN
                did$restore:
                    DO;
                        ddata.restore = FALSE;
                        ddata.status(spindle) = status;
                        IF iors$p <> 0 THEN
                        restart:
                            DO;
                                CALL i206$start(iors$p, ddata$p, duib$p);
                            END restart;
                        RETURN;
                    END did$restore;

                ddata.status(spindle) = status;

                IF iors$p <> 0 THEN
                valid$iors:
                    DO;
                        IF status <> 0 THEN
                        bad$status:
                            DO;
```

```
5                                   iors.status = E$IO;
5                                   IF (status <= 010H) THEN
5                                       temp = status;
5                                   ELSE
                                        temp = shr(status, 4) + 00FH;
5                                   iors.unit$status = unit$status(temp) OR SHL(status,8);
5                                   iors.actual = 0;
5                                   iors.done = TRUE;
5                                   IF need$reset(ddata.status(iors.unit / 4)) THEN
5                                   recalibrate:
                                        DO;
                                            /*
                                             * Note: must index drive select
                                             * bits from iors.unit.
                                             */
6                                           ddata.inter = inter$on$mask;
6                                           ddata.instr = restore$op;
6                                           ddata.restore = send$206$iopb(dinfo.base, @ddata);
6                                       END recalibrate;

5                               END bad$status;
4                           ELSE ok$status:
                                DO;
5                                   iors.actual = iors.count;
5                                   iors.done = TRUE;
5                               END ok$status;
4                   END valid$iors;
3               END done$int;
2           ELSE status$int:
                DO;
3                   temp = input(inter$stat$port);
3                   DO spindle=0 TO 3;
4                       IF (temp AND SHL(1, spindle)) <> 0 THEN
4                           GOTO found$spindle;
4                   END;
3   found$spindle:
                    spindle = SHL(spindle,2);
3                   DO temp=spindle TO spindle+3;
4                       IF ((input(result$byte$port) AND drive$ready(spindle)) = 0) THEN
4                           CALL notify(temp, @ddata);
4                   END;
3               END status$int;

2       END i206$interrupt;
    $subtitle('i206$init')

        /*
         * i206$init
         *      init procedure for the iSBC 206
         *
         * CALLING SEQUENCE:
         *      CALL i206$init(duib$p, ddata$p, status$p);
         *
         * INTERFACE VARIABLES:
         *      duib$p      - pointer to Device-Unit Information Block
         *      ddata$p     - device data segment pointer.
         *      status$p    - pointer to WORD indicating status of operation
         *
         * CALLS:
         *      <none>
         *
         */

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

```
 169   2                DECLARE
                            duib        BASED duib$p DEV$UNIT$INFO$BLOCK,
                            dinfo$p     POINTER,
                            dinfo       BASED dinfo$p I206$DEVICE$INFO,
                            ddata       BASED ddata$p IO$PARM$BLOCK$206,
                            status      BASED status$p WORD;
 170   2                DECLARE
                            i           WORD;

 171   2                dinfo$p = duib.device$info$p;

                        /*
                         * Reset 206;  not there or not hard disk ==> Oops!
                         */

 172   2                output(reset$port) = 0;
 173   2                status = E$OK;

 174   2                ddata.restore = FALSE;

 175   2            END i206$init;

 176   1        END x206ds;


MODULE INFORMATION:

     CODE AREA SIZE     = 0300H    768D
     CONSTANT AREA SIZE = 0034H     52D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0046H     70D
     1037 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     96KB MEMORY AVAILABLE
     18KB MEMORY USED   (18%)
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION


PL/M-86 COMPILER    x206io.p86: iSBC 206 I/O Module
                    Module Header


iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE X206IO
OBJECT MODULE PLACED IN :F1:X206IO.OBJ
COMPILER INVOKED BY:  :LANG:plm86 :F1:X206IO.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE


           $title('x206io.p86: iSBC 206 I/O Module')
           $subtitle('Module Header')
   1       x206io: DO;


           /*
            *             INTEL CORPORATION PROPRIETARY INFORMATION
            *
            *          This software is supplied under the terms of a
            *          license agreement or nondisclosure agreement with
            *          Intel Corporation and may not be copied or disclosed
            *          except in accordance with the terms of that agreement.
            *
            */
```

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

        $include(:f1:xcomon.lit)
    =   $save nolist
        $include(:f1:xnutyp.lit)
    =   $save nolist
        $include(:f1:xiotyp.lit)
    =   $save nolist
        $include(:f1:xparam.lit)
    =   $save nolist
        $include(:f1:x206dv.lit)
    =   $save nolist
        $include(:f1:x206in.lit)
    =   $save nolist
        $include(:f1:xiors.lit)
    =   $save nolist
        $include(:f1:xduib.lit)
    =   $save nolist
        $include(:f1:xtrsec.lit)
    =   $save nolist
        $include(:f1:xexcep.lit)
    =   $save nolist
        $include(:f1:xioexc.lit)
    =   $save nolist
        $include(:f1:x206dc.ext)
    =   $save nolist

        /*
         * this module also does seeks
         */

            DECLARE
                i206$op$codes (*)   BYTE DATA(
                    READ$OP,
                    WRITE$OP,
                    SEEK$OP
                );

        $subtitle('io$206: iSBC 206 I/O Module')

            /*
             * io$206
             *      I/O module (read/write/seek)
             *
             * CALLING SEQUENCE:
             *      CALL io$206 (base, iors$p, duib$p, iopb$p);
             *
             * INTERFACE VARIABLES:
             *      base        - base address of the board.
             *      iors$p      - I/O Request/Result segment pointer
             *      duib$p      - pointer to Device-Unit Information Block
             *      iopb$p      - pointer to I/O parameter block.
             *
             * CALLS:
             *      send$206$iopb(base, @iopb)
             */
            io$206: PROCEDURE (base, iors$p, duib$p, iopb$p) REENTRANT PUBLIC;
                DECLARE
                    base        WORD,
                    iors$p      POINTER,
                    duib$p      POINTER,
                    iopb$p      POINTER;
```

```
 2              DECLARE
                    iors        BASED iors$p IO$REQ$RES$SEG,
                    ts          DWORD,
                    ts$o        TRACK$SECTOR$STRUCT AT(@ts),
                    duib        BASED duib$p DEV$UNIT$INFO$BLOCK,
                    iopb        BASED iopb$p IO$PARM$BLOCK$206,
                    platter     BYTE,
                    spindle     BYTE,
                    surface     BYTE;

 2              ts = iors.dev$loc;

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

 2              iopb.inter = INTER$ON$MASK;
 2              iopb.cyl$add = shr(ts$o.track, 1);            /* track/2 = cylinder */
 2              iopb.instr = i206$op$codes(iors.funct)       OR
                             shl(spindle, 4) OR
                             shl(platter, 6) OR
                             shl(surface, 3);

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

 2              iopb.r$count = iors.count / duib.dev$gran;      /* divide by sectors size */
 2              iopb.rec$add = (ts$o.sector + 1) OR
                                shr(ts$o.track AND 0200H, 2);   /* (cyl AND 0100H) / 2 */
 2              iopb.buff$p = iors.buff$p;

 2              IF NOT send$206$iopb(base, @iopb) THEN

                    /*
                     * the board did not accept the iopb so...
                     */
 2                  DO;
 3                      iors.status = IO$SOFT;
 3                      iors.actual = 0;
 3                      iors.done = TRUE;
 3                  END;

 2          END io$206;

 1      END x206io;


MODULE INFORMATION:

     CODE AREA SIZE     = 00D5H    213D
     CONSTANT AREA SIZE = 0003H      3D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0022H     34D
     605 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     96KB MEMORY AVAILABLE
     12KB MEMORY USED   (12%)
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER    x206dc: iSBC 206 parameter handler
                    Module Header


iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE X206DC
OBJECT MODULE PLACED IN :F1:X206DC.OBJ
COMPILER INVOKED BY:  :LANG:plm86 :F1:X206DC.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE


              $title('x206dc: iSBC 206 parameter handler')
              $subtitle('Module Header')
   1          x206dc: DO;


              /*
               *              INTEL CORPORATION PROPRIETARY INFORMATION
               *
               *           This software is supplied under the terms of a
               *           license agreement or nondisclosure agreement with
               *           Intel Corporation and may not be copied or disclosed
               *           except in accordance with the terms of that agreement.
               *
               */

              /*
               *  This module contains the commands for the 206 controller.
               *
               * LANGUAGE DEPENDENCIES: COMPACT ROM OPTIMIZE(3)
               */

              $include(:f1:xcomon.lit)
           =  $save nolist
              $include(:f1:xnutyp.lit)
           =  $save nolist
              $include(:f1:x206dv.lit)
           =  $save nolist

              $subtitle('Send 206 I/O Parameter Block')

                 /*
                  * send$206$iopb
                  *      send the iSBC 206 the address of the parameter block
                  *
                  * CALLING SEQUENCE:
                  *      CALL send$206$iopb (base, iopb$p);
                  *
                  * INTERFACE VARIABLES:
                  *      base        - base address of board.
                  *      iopb$p      - I/O parameter block pointer
                  *
                  * CALLS:
                  *      (none)
                  */
   9   1         send$206$iopb: PROCEDURE (base, iopb$p) BOOLEAN REENTRANT PUBLIC;
  10   2            DECLARE
                        base        WORD,
                        iopb$p      POINTER;
  11   2            DECLARE
                        iopb$p$o    P$OVERLAY AT(@iopb$p),
                        iopb        BASED iopb$p IO$PARM$BLOCK$206,
                        drive       BYTE;

  12   2            drive = shr(iopb.instr AND 030H, 4);
  13   2            drive = shl(01H,drive);

  14   2            IF (input(controller$stat)) = (COMMAND$BUSY OR drive) THEN
  15   2                RETURN(FALSE);

  16   2            output (io$off$port) = low (iopb$p$o.offset);
```

```
 17   2             IF (input(controller$stat) AND COMMAND$BUSY) <> 0 THEN
 18   2                RETURN(FALSE);

                    /*
                     * made it to here so output rest of iopb address
                     */

 19   2             output (lo$seg$port) = low (iopb$p$o.base);
 20   2             output (hi$seg$port) = high (iopb$p$o.base);
 21   2             output (hi$off$port) = high (iopb$p$o.offset);

 22   2             RETURN(TRUE);

 23   2          END send$206$iopb;

 24   1       END x206dc;


MODULE INFORMATION:

     CODE AREA SIZE     = 0060H     96D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 000CH     12D
     208 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     96KB MEMORY AVAILABLE
     5KB MEMORY USED   (5%)
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION



 PL/M-86 COMPILER    x206fm.p86
                     Module Header


 iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE X206FM
 OBJECT MODULE PLACED IN :F1:X206FM.OBJ
 COMPILER INVOKED BY:  :LANG:plm86 :F1:X206FM.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE


               $title('x206fm.p86')
               $subtitle('Module Header')

               /*
                *  x206fm.p86
                *
                *  iSBC 206 device
                *      formats one track on hard disk
                *
                * LANGUAGE DEPENDENCIES: COMPACT ROM OPTIMIZE(3)
                */

   1           x206fm: DO;
```

```
    /*
     *              INTEL CORPORATION PROPRIETARY INFORMATION
     *
     *          This software is supplied under the terms of a
     *          license agreement or nondisclosure agreement with
     *          Intel Corporation and may not be copied or disclosed
     *          except in accordance with the terms of that agreement.
     *
     */

    /*
     *  This module builds the 206 parameter block and passes the
     *      address of it to the iSBC 206.
     *
     *      note: this format procedure deduces max$sectors from the
     *      DEV$UNIT$INFO$BLOCK (duib.dev$gran).  it does NOT check to see if
     *      the operator has set the switches on the controller correctly.
     *      sectors may be 128 or 512 bytes.
     *
     */

    $include(:f1:xcomon.lit)
=   $save nolist
    $include(:f1:xnutyp.lit)
=   $save nolist
    $include(:f1:xiotyp.lit)
=   $save nolist
    $include(:f1:xparam.lit)
=   $save nolist
    $include(:f1:x206dv.lit)
=   $save nolist
    $include(:f1:x206in.lit)
=   $save nolist
    $include(:f1:xradsf.lit)
=   $save nolist
    $include(:f1:xiors.lit)
=   $save nolist
    $include(:f1:xduib.lit)
=   $save nolist
    $include(:f1:xtrsec.lit)
=   $save nolist
    $include(:f1:xexcep.lit)
=   $save nolist
    $include(:f1:xioexc.lit)
=   $save nolist

    $include(:f1:x206dc.ext)
=   $save nolist

    $subtitle('format$206: Format track procedure')

        /*
         * format$206
         *      format a track on the 206
         *
         * CALLING SEQUENCE:
         *      CALL format$206 (base, iors$p, duib$p, iopb$p);
         *
         * INTERFACE VARIABLES:
         *      base        - base address of board.
         *      iors$p      - I/O Request/Result segment pointer
         *      duib$p      - pointer to Device-Unit Information Block
         *      iopb$p      - I/O parameter block pointer.
         *
         * CALLS:
         *      build$206$fmt$table
         *      send$206$iopb
         */
```

```
36   1         format$206: PROCEDURE (base, iors$p, duib$p, iopb$p) REENTRANT PUBLIC;
37   2            DECLARE
                     base            WORD,
                     iors$p          POINTER,
                     duib$p          POINTER,
                     iopb$p          POINTER;
38   2            DECLARE
                     iors        BASED iors$p IO$REQ$RES$SEG,
                     format$info$p   POINTER,
                     format$info BASED format$info$p FORMAT$INFO$STRUCT,
                     duib        BASED duib$p DEV$UNIT$INFO$BLOCK,
                     iopb        BASED iopb$p IO$PARM$BLOCK$206,
                     platter             BYTE,
                     spindle             BYTE,
                     surface             BYTE,
                     max$sectors         BYTE;

39   2            format$info$p = iors.aux$p;
40   2            IF format$info.track$num > i206$TRACK$MAX THEN
41   2               DO;
42   3                  iors.status = E$SPACE;
43   3                  iors.actual = 0;
44   3                  iors.done = TRUE;
45   3                  RETURN;
46   3               END;

47   2            spindle = shr(iors.unit, 2);                       /* 4 units/spindle */
48   2            platter = iors.unit AND 003H;                      /* (as above ) */
49   2            surface = format$info.track$num AND 00001H;        /* select surface */

50   2            iopb.inter = INTER$ON$MASK OR FORMAT$TRACK$ON;
51   2            iopb.cyl$add = shr(format$info.track$num, 1);    /* track/2 = cylinder */
52   2            iopb.rec$add = shr(format$info.track$num AND 0200H, 2); /* set if over 256 cylinders */
53   2            iopb.instr = format$op          OR
                               shl(spindle, 4) OR
                               shl(platter, 6) OR
                               shl(surface, 3);

54   2            iopb.buff$p = @iopb.format$table;

55   2            IF duib.dev$gran = 128 THEN
56   2               max$sectors = 36;
57   2            ELSE
                     /*
                      * if not 128 then MUST be 512 byte sectors
                      */
                     max$sectors = 12;

58   2            CALL build$206$fmt$table(@iopb.format$table,
                                           format$info.track$num,
                                           format$info.track$interleave,
                                           format$info.track$skew,
                                           format$info.fill$char,
                                           max$sectors);

59   2            IF NOT send$206$iopb(base, @iopb) THEN
                     /*
                      * the board did not accept the iopb so...
                      */
60   2               DO;
61   3                  iors.status = IO$SOFT;
62   3                  iors.actual = 0;
63   3                  iors.done = TRUE;
64   3               END;

65   2         END format$206;
```

```
        /*
         * build$206$fmt$table
         *      fill out format table
         *
         * CALLING SEQUENCE:
         *      CALL build$206$fmt$table(buf$p, track, int$fact, skew, fill$char, max$sectors);
         *
         * INTERFACE VARIABLES:
         *      buf$p       - address of format table.
         *      track       - track to be formatted.
         *      int$fact    - interleave factor.
         *      skew        - squew from physical  sector one.
         *      fill$char   - used to fill sectors.
         *      max$sectors - maximum number of sectors
         *
         * CALLS:
         *      <none>
         *
         * No error checking on skew, int$fact parameters; if nonsense, the algorithm
         * completes & formats the track in a strange manner.
         */

        build$206$fmt$table: PROCEDURE(buf$p, track, int$fact, skew, fill$char,max$sectors) REENTRANT;
            DECLARE
                buf$p       POINTER,
                track       WORD,
                int$fact    BYTE,
                skew        BYTE,
                fill$char   BYTE,
                max$sectors BYTE;
            DECLARE
                s           BYTE,
                i           BYTE;
            DECLARE
                fmt$tab BASED buf$p (36) STRUCTURE(
                    record$address  BYTE,
                    fill$char       BYTE);

            DO i = 0 TO (max$sectors - 1);
                fmt$tab(i).record$address = 0FFH;
                fmt$tab(i).fill$char = fill$char;
            END;

            s = skew MOD max$sectors;

            DO i = 1 TO max$sectors;

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

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

        END build$206$fmt$table;

    END x206fm;

            MODULE INFORMATION:

                 CODE AREA SIZE     = 0192H    402D
                 CONSTANT AREA SIZE = 0000H      0D
                 VARIABLE AREA SIZE = 0000H      0D
                 MAXIMUM STACK SIZE = 0028H     40D
                 743 LINES READ
                 0 PROGRAM WARNINGS
                 0 PROGRAM ERRORS

            DICTIONARY SUMMARY:

                 96KB MEMORY AVAILABLE
                 13KB MEMORY USED   (13%)
                 0KB DISK SPACE USED

            END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER    x8274: 8274 terminal device driver
                    Module Header


iRMX 86 PL/M-86 V2.3 COMPILATION OF MODULE X8274
OBJECT MODULE PLACED IN :F1:X8274.OBJ
COMPILER INVOKED BY:  :LANG:plm86 :F1:X8274.P86 COMPACT OPTIMIZE(3) ROM PAGEWIDTH(132) NOTYPE


               $title('x8274: 8274 terminal device driver')
               $subtitle('Module Header')

               /*
                * TITLE:       x8274
                *
                * DATE:        27 FEB 84
                *
                * ABSTRACT:    This module is the interface between the iRMX 86
                *              Terminal Support, and the 8274 MPSC.  It is a
                *              rewritten version of a module of the same name dated
                *              20 Jan 83.  The rewritting was necessary to correct
                *              initialization timing problems and to add support for
                *              various timer devices, i.e. 8253-4, 80130, and 80186-8.
                *
                * LANGUAGE DEPENDENCIES:   PLM86 COMPACT ROM
                */

               /*
                *              INTEL CORPORATION PROPRIETARY INFORMATION
                *
                *          This software is supplied under the terms of a
                *          license agreement or nondisclosure agreement with
                *          Intel Corporation and may not be copied or disclosed
                *          except in accordance with the terms of that agreement.
                *
                */

   1            x8274: DO;

               $include(:f1:xcomon.lit)
           =   $save nolist
               $include(:f1:xnutyp.lit)
           =   $save nolist
               $include(:f1:xiotyp.lit)
           =   $save nolist
               $include(:f1:xexcep.lit)
           =   $save nolist

               $include(:f1:xtssow.ext)
           =   $save nolist
               $include(:f1:xtstim.ext)
           =   /*
           =    * External Declaration
           =    * for timer support procedure.
           =    */
           =   $SAVE NOLIST
               $include(:f1:xdelay.ext)
           =   $SAVE NOLIST

               $subtitle('Data structures and literals')

               /*
                *  8274 register values
                */
```

```
20   1      DECLARE
                WR0                         LITERALLY '00H',
                WR1                         LITERALLY '01H',
                WR2                         LITERALLY '02H',
                WR3                         LITERALLY '03H',
                WR4                         LITERALLY '04H',
                WR5                         LITERALLY '05H',
                WR6                         LITERALLY '06H',
                WR7                         LITERALLY '07H',
                RR0                         LITERALLY '00H',
                RR1                         LITERALLY '01H',
                RR2                         LITERALLY '02H';

            /*
             *  8274 command values
             */

21   1      DECLARE
                NULL_CMD                    LITERALLY '00H',
                NULL_VECTOR                 LITERALLY '00H',
                RESET_EXT_INT               LITERALLY '10H',
                CHANNEL_RESET               LITERALLY '18H',
                ENABLE_INT_NEXT_RX          LITERALLY '20H',
                RESET_TX_INT                LITERALLY '28H',
                ERROR_RESET                 LITERALLY '30H',
                END_OF_INT                  LITERALLY '38H';

            /*
             *  8274 write register commands.
             */

22   1      DECLARE
                WR1_INIT                    LITERALLY '016H',   /* int on all Rx chars and
                                                                 * special conditions.
                                                                 * Parity affects vector,
                                                                 * variable vector,
                                                                 * Tx int enable, No
                                                                 * external int.
                                                                 */
                WR1_NO_RX_INT               LITERALLY '006H',   /* disable Rx interrupts */
                WR1_NO_INT                  LITERALLY '004H',   /* Disable Rx and Tx
                                                                 * interrupts
                                                                 */
                WR2_INIT                    LITERALLY '004H',   /* non vectored int */
                WR3_INIT                    LITERALLY '0C1H',   /* Rx 8 bits/char,
                                                                 * Rx enable
                                                                 */
                WR3_RX_DISABLE              LITERALLY '0C0H',
                WR4_INIT                    LITERALLY '044H',   /* 16X clock, 8 bit data,
                                                                 * 1 stop bit, no parity
                                                                 */
                WR5_TX_ENABLE               LITERALLY '0EAH',   /* Tx 8 bits data,
                                                                 * Tx enable, RTS enable
                                                                 */
                WR5_TX_DISABLE              LITERALLY '0E2H',
                WR5_DTR_ON                  LITERALLY '0EAH',
                WR5_DTR_OFF                 LITERALLY '06AH';

            /*
             *  status register bit masks
             */
23   1      DECLARE
                VECTOR_MASK             LITERALLY '0E0H',
                TEST_VECTOR             LITERALLY '0A5H',
                INT_PENDING             LITERALLY '002H',
                NO_INT_VECT             LITERALLY '01CH',
                RX_CHAR_RDY             LITERALLY '001H',
                TX_BUFFER_EMPTY         LITERALLY '004H',
                i8274$INPUT$ERROR       LITERALLY '070H';
```

```
                 /*
                  *  Flags values
                  */

24    1          DECLARE
                     EVEN$MODE                LITERALLY '003H',
                     ODD$MODE                 LITERALLY '001H',
                     NO$PARITY$MODE           LITERALLY '000H',
                     IN$PARITY$MASK           LITERALLY '030H',
                     OUT$PARITY$MASK          LITERALLY '1C0H',
                     STRIP$INPUT$PARITY$MODE  LITERALLY '000H',
                     PASS$INPUT$PARITY$MODE   LITERALLY '010H',
                     EVEN$INPUT$PARITY$MODE   LITERALLY '020H',
                     ODD$INPUT$PARITY$MODE    LITERALLY '030H',
                     SPACE$OUTPUT$PARITY$MODE    LITERALLY '000H',
                     MARK$OUTPUT$PARITY$MODE  LITERALLY '040H',
                     EVEN$OUTPUT$PARITY$MODE  LITERALLY '080H',
                     ODD$OUTPUT$PARITY$MODE   LITERALLY '0C0H',
                     PASS$OUTPUT$PARITY$MODE  LITERALLY '100H',
                     OUT$PAR$CHECK            LITERALLY '080H';

                 /*
                  *  Baud rate values
                  */

25    1          DECLARE
                     HARDWARE$BAUD$SELECT     LITERALLY '0',
                     AUTO$BAUD$SELECT         LITERALLY '3',
                     OUT$BAUD$SAME            LITERALLY '1';

                 /*
                  *  interface to terminal support
                  */

26    1          DECLARE
                     MORE$INTERRUPT           LITERALLY '08H',
                     NO$INTERRUPT             LITERALLY '00H',
                     DELAY$INTERRUPT          LITERALLY '05H + MORE$INTERRUPT',
                     INPUT$INTERRUPT          LITERALLY '01H + MORE$INTERRUPT',
                     OUTPUT$INTERRUPT         LITERALLY '02H + MORE$INTERRUPT',
                     RING$INTERRUPT           LITERALLY '03H + MORE$INTERRUPT',
                     CARRIER$INTERRUPT        LITERALLY '04H + MORE$INTERRUPT';


                 /*
                  * Controller Data Structure
                  */

27    1          DECLARE
                     TS$CDATA     LITERALLY     'STRUCTURE(
                                                           TS$CDATA1,
                                                           TS$CDATA2)';

28    1          DECLARE
                     TS$CDATA1    LITERALLY    'ics$data$segment        SEGMENT,
                                                status                  WORD,
                                                interrupt$type          BYTE,
                                                interrupting$unit       BYTE,
                                                dinfo$p                 POINTER,
                                                driver$cdata$p          POINTER',
                     TS$CDATA2    LITERALLY    'reserved(34)            BYTE,
                                                udata(1)                BYTE';
```

```
             /*
              * Unit Data Structure
              */

29    1      DECLARE
                 TS$UDATA     LITERALLY    'STRUCTURE(
                                                 TS$UDATA1,
                                                 TS$UDATA2,
                                                 TS$UDATA3)';

30    1      DECLARE
                 TS$UDATA1    LITERALLY    'uinfo$p                POINTER,
                                            term$flags             WORD,
                                            in$rate                WORD,
                                            out$rate               WORD,
                                            scroll$number          WORD,
                                            translation(87)        BYTE',

                 TS$UDATA2    LITERALLY    'input$control$table(33)    BYTE,
                                            unit$number            BYTE',

                 TS$UDATA3    LITERALLY    'fill(891)              BYTE';

             /*
              *  8274 Device information Structure
              */

31    1      DECLARE
                 i8274$CONTROLLER$INFO   LITERALLY   'STRUCTURE(
                                                         i8274$INFO$1,
                                                         i8274$INFO$2,
                                                         i8274$INFO$3,
                                                         i8274$INFO$4,
                                                         i8274$INFO$5,
                                                         i8274$INFO$6,
                                                         i8274$INFO$7)';

32    1      DECLARE
                 i8274$INFO$1     LITERALLY    'filler(12)          WORD',
                 i8274$INFO$2     LITERALLY    'ch_a_data_port      WORD,
                                                ch_a_status_port    WORD,
                                                ch_b_data_port      WORD,
                                                ch_b_status_port    WORD',
                 i8274$INFO$3     LITERALLY    'ch_a_in_rate_port   WORD,
                                                ch_a_in_rate_cmd_port WORD,
                                                ch_a_in_rate_counter   BYTE,
                                                ch_a_in_rate_freq   DWORD',
                 i8274$INFO$4     LITERALLY    'ch_a_out_rate_port WORD,
                                                ch_a_out_rate_cmd_port WORD,
                                                ch_a_out_rate_counter  BYTE,
                                                ch_a_out_rate_freq DWORD',
                 i8274$INFO$5     LITERALLY    'ch_b_in_rate_port   WORD,
                                                ch_b_in_rate_cmd_port  WORD,
                                                ch_b_in_rate_counter   BYTE,
                                                ch_b_in_rate_freq   DWORD',
                 i8274$INFO$6     LITERALLY    'ch_b_out_rate_port WORD,
                                                ch_b_out_rate_cmd_port WORD,
                                                ch_b_out_rate_counter  BYTE,
                                                ch_b_out_rate_freq DWORD',
                 i8274$INFO$7     LITERALLY    'ch_a_timer_type     BYTE,
                                                ch_b_timer_type     BYTE';
```

```
$subtitle('i8274$init')

/*
 *  TITLE:  i8274$init
 *
 *  CALLING SEQUENCE:
 *      CALL i8274$init(cdata$p);
 *
 *  INTERFACE VARIABLES:
 *      cdata$p         POINTER to controller data
 *
 *  CALLS:
 *      none
 *
 *  ABSTRACT:
 *      Initializes the 8274 chip.
 */

i8274$init: PROCEDURE(cdata$p) REENTRANT PUBLIC;

DECLARE
    cdata$p                             POINTER,
    cdata   BASED   cdata$p             TS$CDATA;

DECLARE
    i8274$info$p                        POINTER,
    i8274$info  BASED   i8274$info$p    i8274$CONTROLLER$INFO;

DECLARE
    port                                WORD;


/*
 *      Get the configuration info
 */

    i8274$info$p = cdata.dinfo$p;

/*
 *  Initialize driver data area (10 bytes in length)
 */

    CALL setb(0FFH, cdata.driver$cdata$p, 10);

/*
 *  Reset and Initialize the 8274.
 */

    DISABLE;
    port = i8274$info.ch_a_status_port;

    OUTPUT(port) = WR0;             /* point to WR0 */
    CALL delay(10);                 /* insure delay between outputs */
    OUTPUT(port) = CHANNEL_RESET;   /* reset channel A */
    CALL delay(10);                 /* insure delay between outputs */
    ENABLE;

    DISABLE;
    port = i8274$info.ch_b_status_port;

    OUTPUT(port) = WR0;             /* point to WR0 */
    CALL delay(10);                 /* insure delay between outputs */
    OUTPUT(port) = CHANNEL_RESET;   /* reset channel A */
    CALL delay(10);                 /* insure delay between outputs */
    ENABLE;
```

```
  2          DISABLE;
  2          OUTPUT(port) = WR4;                /* point to WR4 */
  2          CALL delay(10);                    /* insure delay between outputs */
  2          OUTPUT(port) = WR4_INIT;           /* initialize WR4 */
  2          CALL delay(10);                    /* insure delay between outputs */
  2          ENABLE;

  2          DISABLE;
  2          OUTPUT(port) = WR5;                /* point to WR5 */
  2          CALL delay(10);                    /* insure delay between outputs */
  2          OUTPUT(port) = WR5_TX_ENABLE;      /* initialize WR5 - Tx enabled */
  2          CALL delay(10);                    /* insure delay between outputs */
  2          ENABLE;

  2          DISABLE;
  2          OUTPUT(port) = WR3;                /* point to WR3 */
  2          CALL delay(10);                    /* insure delay between outputs */
  2          OUTPUT(port) = WR3_INIT;           /* initialize WR3 - Rx enabled */
  2          CALL delay(10);                    /* insure delay between outputs */
  2          ENABLE;

  2          DISABLE;
  2          OUTPUT(port) = WR1;                /* point to WR1 */
  2          CALL delay(10);                    /* insure delay between outputs */
  2          OUTPUT(port) = WR1_NO_INT;         /* initialize WR1 - Interrupts disabled */
  2          CALL delay(10);                    /* insure delay between outputs */
  2          ENABLE;

  2          DISABLE;
  2          port = i8274$info.ch_a_status_port;

  2          OUTPUT(port) = WR4;                /* point to WR4 */
  2          CALL delay(10);                    /* insure delay between OUTPUTs */
  2          OUTPUT(port) = WR4_INIT;           /* initialize WR4 */
  2          CALL delay(10);                    /* insure delay between OUTPUTs */
  2          ENABLE;

  2          DISABLE;
  2          OUTPUT(port) = WR5;                /* point to WR5 */
  2          CALL delay(10);                    /* insure delay between OUTPUTs */
  2          OUTPUT(port) = WR5_TX_ENABLE;      /* initialize WR5 - Tx enabled */
  2          CALL delay(10);                    /* insure delay between OUTPUTs */
  2          ENABLE;

  2          DISABLE;
  2          OUTPUT(port) = WR3;                /* point to WR3 */
  2          CALL delay(10);                    /* insure delay between OUTPUTs */
  2          OUTPUT(port) = WR3_INIT;           /* initialize WR3 - Rx enabled */
  2          CALL delay(10);                    /* insure delay between OUTPUTs */
  2          ENABLE;

  2          DISABLE;
  2          OUTPUT(port) = WR1;                /* point to WR1 */
  2          CALL delay(10);                    /* insure delay between OUTPUTs */
  2          OUTPUT(port) = WR1_NO_INT;         /* initialize WR1 - Interrupts disabled */
  2          CALL delay(10);                    /* insure delay between OUTPUTs */
  2          ENABLE;

  2          DISABLE;
  2          port = i8274$info.ch_a_status_port;

  2          OUTPUT(port) = WR2;                /* point to WR2 */
  2          CALL delay(10);                    /* insure delay between OUTPUTs */
  2          OUTPUT(port) = WR2_INIT;           /* initialize WR2 - non vectored int */
  2          CALL delay(10);                    /* insure delay between OUTPUTs */
  2          ENABLE;
```

```
        DISABLE;
        port = i8274$info.ch_b_status_port;

        OUTPUT(port) = WR2;             /* point to WR2 */
        CALL delay(10);                 /* insure delay between OUTPUTs */
        OUTPUT(port) = NULL_VECTOR;     /* initialize WR2 - non vectored int */
        ENABLE;

    /*
     * Set the interrrupt vector in R2B to some value, and then read it
     * back to see if the chip is really there, then set to the desired
     * value.
     */
        cdata.status = E$OK;

        OUTPUT(port) = WR2;             /* point to WR2 */
        CALL delay(10);                 /* insure delay between OUTPUTs */
        OUTPUT(port) = TEST_VECTOR;     /* interrupt vector for RR2B */

        CALL TIME(10);

        OUTPUT(port) = RR2;             /* point to RR2 */
        CALL delay(10);                 /* insure delay between OUTPUTs */

        IF (INPUT(port) AND VECTOR_MASK) <> (TEST_VECTOR AND VECTOR_MASK)
        THEN
            cdata.status = E$IO;

        CALL TIME(10);

        OUTPUT(port) = WR2;             /* point to WR2 */
        CALL delay(10);                 /* insure delay between OUTPUTs */
        OUTPUT(port) = NULL_VECTOR;     /* null interrupt vector for RR2B */

        CALL TIME(10);

        OUTPUT(port) = RR2;             /* point to RR2 */
        CALL delay(10);                 /* insure delay between OUTPUTs */

        IF (INPUT(port) AND VECTOR_MASK) <> 0
        THEN
            cdata.status = E$IO;

    END i8274$init;

    $subtitle('i8274$setup')

    /*
     *  TITLE:  i8274$setup
     *
     *  CALLING SEQUENCE:
     *      CALL i8274$setup(udata$p);
     *
     *  INTERFACE VARIABLES:
     *      udata$p     POINTER to unit data
     *
     *  CALLS:
     *      none
     *
     *  ABSTRACT:
     *      Initializes the baud rate generator to the configured
     *      rate, and sets up the 8274 for asychronous mode,
     *      divide by 16, 8 data bits, 1 stop
     *      bit; parity generation per configuration.
     */
```

```
i8274$setup: PROCEDURE(udata$p) REENTRANT PUBLIC;

DECLARE
    udata$p                                 POINTER,
    udata$p$o       STRUCTURE(
                                  offset    WORD,
                                  base      SELECTOR) AT(@udata$p),
    cdata   BASED   udata$p$o.base          TS$CDATA,
    udata   BASED   udata$p                 TS$UDATA;

DECLARE
    i8274$info$p                            POINTER,
    i8274$info  BASED   i8274$info$p        i8274$CONTROLLER$INFO;

DECLARE
    ch_p                                    POINTER,
    ch  BASED   ch_p    STRUCTURE (
                                    data_port       WORD,
                                    status_port     WORD ),
    ch_rate_p                               POINTER,
    ch_rate BASED   ch_rate_p   STRUCTURE (
                                    in_port         WORD,
                                    in_cmd_port     WORD,
                                    in_counter      BYTE,
                                    in_freq         DWORD,
                                    out_port        WORD,
                                    out_cmd_port    WORD,
                                    out_counter     BYTE,
                                    out_freq        DWORD),

    driver$data$p                           POINTER,
    driver$data BASED   driver$data$p   STRUCTURE(
                                    ch_a$in$rate    WORD,
                                    ch_a$out$rate   WORD,
                                    ch_a$parity     BYTE,
                                    ch_b$in$rate    WORD,
                                    ch_b$out$rate   WORD,
                                    ch_b$parity     BYTE);


DECLARE
    temp                                    BYTE,
    port                                    WORD,
    out_cmd                                 BYTE,
    parity$mode                             BYTE,
    timer$type                              BYTE,
    rate$count                              WORD,
    in$rate                                 WORD,
    out$rate                                WORD,
    parity                                  BYTE;


    i8274$info$p = cdata.dinfo$p;
    driver$data$p = cdata.driver$cdata$p;

    IF udata.unit$number = 0 THEN
    DO;
        ch_p = @i8274$info.ch_a_data_port;
        ch_rate_p = @i8274$info.ch_a_in_rate_port;
        timer$type = i8274$info.ch_a_timer_type;
        in$rate = driver$data.ch_a$in$rate;
        out$rate = driver$data.ch_a$out$rate;
        parity = driver$data.ch_a$parity;
    END;
    ELSE
```

```
        DO;
            ch_p = @i8274$info.ch_b_data_port;
            ch_rate_p = @i8274$info.ch_b_in_rate_port;
            timer$type = i8274$info.ch_b_timer_type;
            in$rate = driver$data.ch_b$in$rate;
            out$rate = driver$data.ch_b$out$rate;
            parity = driver$data.ch_b$parity;
        END;

        out_cmd = WR5_TX_ENABLE;

    /*
     *  Initialize the input rate generator if the baud rate has changed, or
     *  if a baud rate scan is in progress, and if it's programmable.
     */

        IF (in$rate <> udata.in$rate) AND
            (ch_rate.in_freq <> 0) AND (udata.in$rate <> HARDWARE$BAUD$SELECT)
            THEN
        DO;

            IF udata.in$rate <= AUTO$BAUD$SELECT THEN
            DO;
                rate$count = SHR(19200, (udata.in$rate-1)*3);
                out_cmd = WR5_TX_DISABLE;
            END;
            ELSE
            DO;
                rate$count = udata.in$rate;
                in$rate = udata.in$rate;
            END;

           /*
            * The initial timer value is the timer input frequency
            * divided by the configured baud rate.
            */

            temp = FALSE;
            IF (ch_rate.in_freq MOD rate$count) >= SHR(rate$count,1) THEN
                temp = TRUE;
            rate$count = (ch_rate.in_freq / rate$count);
            IF temp THEN
                rate$count = rate$count + 1;

            CALL set$baud$rate$count(ch_rate.in_cmd_port,
                                     ch_rate.in_port,
                                     timer$type,
                                     ch_rate.in_counter,
                                     rate$count);

        END;

    /*
     *  initialize the output baud rate generator, if there is one, and it has
     *  changed, and it's programmable.
     */

        IF (out$rate <> udata.out$rate) AND
            (ch_rate.out_freq <> 0) AND (udata.out$rate <> HARDWARE$BAUD$SELECT)
            THEN
        DO;
            IF udata.out$rate <> OUT$BAUD$SAME THEN
            DO;
                temp = FALSE;
                IF (ch_rate.out_freq MOD udata.out$rate) >= SHR(udata.out$rate,1)
                     THEN
                        temp = TRUE;
                rate$count = (ch_rate.out_freq / udata.out$rate);
                IF temp THEN
                    rate$count = rate$count + 1;
            END;

            out$rate = udata.out$rate;

           /*
            * The initial timer value is the timer output frequency
            * divided by the configured baud rate.
            */

            CALL set$baud$rate$count(ch_rate.out_cmd_port,
                                     ch_rate.out_port,
                                     timer$type,
                                     ch_rate.out_counter,
                                     rate$count);

        END;


    /*
     *  figure out the parity control part of the mode word.
     */

        IF (udata.term$flags AND OUT$PARITY$MASK) = EVEN$OUTPUT$PARITY$MODE THEN
            parity$mode = EVEN$MODE;
        ELSE
        DO;

            IF (udata.term$flags AND OUT$PARITY$MASK) = ODD$OUTPUT$PARITY$MODE
            THEN
                parity$mode = ODD$MODE;
            ELSE
                parity$mode = NO$PARITY$MODE;

        END;


        port = ch.status_port;

        /*
         *  If a new parity is specified, set up this 8274 channel accordingly.
         */

        IF parity$mode <> parity THEN
        DO;
            parity = parity$mode;

            OUTPUT(port) = WR4;                             /* point to WR4 */
            CALL delay(10);                  /* insure delay between OUTPUTs */
            OUTPUT(port) = WR4_INIT OR parity$mode;

            CALL TIME(10);
        END;

        OUTPUT(port) = WR5;                             /* point to WR5 */
        CALL delay(10);                  /* insure delay between outputs */
        OUTPUT(port) = out_cmd;

        CALL TIME(10);

        OUTPUT(port) = WR3;                             /* point to WR3 */
        CALL delay(10);                  /* insure delay between outputs */
        OUTPUT(port) = WR3_INIT;
        CALL delay(10);                  /* insure delay between outputs */
```

```
          /*
           * Throw away any chars from baud rate search.
           */
   2         DO WHILE (INPUT(ch.status_port) AND RX_CHAR_RDY) <> 0;
   3            temp = INPUT(ch.data_port);
   3            CALL delay(10);               /* insure delay between outputs */
   3         END;

          /*
           * If the 8274 is ready for output, tell the terminal support
           * to send a char.
           */

   2         CALL delay(10);               /* insure delay between outputs */
   2         IF (INPUT(ch.status_port) AND TX_BUFFER_EMPTY) <> 0 THEN
   2            CALL xts$set$output$waiting(udata$p);

          /*
           * Allow Tx and Rx interrupts now.
           */

   2         CALL delay(10);               /* insure delay between outputs */
   2         OUTPUT(port) = WR1;                        /* point to WR1 */
   2         CALL delay(10);               /* insure delay between outputs */
   2         OUTPUT(port) = WR1_INIT;


   2      END i8274$setup;

          $subtitle('i8274$check')

          /*
           *  TITLE:  i8274$check
           *
           *  CALLS:
           *     none
           *
           *  INTERFACE VARIABLES:
           *     cdata$p         POINTER to controller data
           *
           *  CALLING SEQUENCE:
           *     ch = i8274$check(cdata$p);
           *
           *  ABSTRACT:
           *     Term$check procedure, connected to 8274 input interrupt.
           *     Gets input char, strips off parity if required, and sets
           *     up flags for terminal support.
           */

   1      i8274$check:   PROCEDURE(cdata$p) BYTE REENTRANT PUBLIC;

   2      DECLARE
             cdata$p                        POINTER,
             cdata   BASED   cdata$p        TS$CDATA;

   2      DECLARE
             i8274$info$p                   POINTER,
             i8274$info  BASED   i8274$info$p    i8274$CONTROLLER$INFO,
             udata$p                        POINTER,
             udata$p$o        STRUCTURE (
                                     offset  WORD,
                                     base    SELECTOR ) AT (@udata$p),
             udata   BASED   udata$p TS$UDATA;
```

```
DECLARE
    ch_p                                POINTER,
    ch  BASED   ch_p    STRUCTURE (
                             data_port    WORD,
                             status_port  WORD ),
    ch_rate_p                           POINTER,
    ch_rate BASED   ch_rate_p   STRUCTURE (
                             in_port      WORD,
                             in_cmd_port  WORD,
                             in_counter   BYTE,
                             in_freq      DWORD,
                             out_port     WORD,
                             out_cmd_port    WORD,
                             out_counter  BYTE,
                             out_freq     DWORD );

DECLARE
    unit                                BYTE,
    vector                              BYTE,
    dummy                               BYTE,
    found$rate                          BYTE,
    i                                   WORD,
    char                                BYTE;

    i8274$info$p = cdata.dinfo$p;

/*
 *  find out what caused the interrupt by reading RR2B
 */
    OUTPUT(i8274$info.ch_b_status_port) = 002H;
    CALL delay(5);                  /* insure delay between outputs */
    vector = INPUT(i8274$info.ch_b_status_port);
    CALL delay(20);                 /* insure delay between outputs */

    IF ((vector AND NO_INT_VECT) = NO_INT_VECT) AND
       ((INPUT(i8274$info.ch_a_status_port) AND INT_PENDING) = 0) THEN
    DO;
        c$data.interrupt$type = NO$INTERRUPT;
        RETURN char;
    END;

    IF (vector AND 10H) = 10H THEN
    DO;
        ch_p = @i8274$info.ch_a_data_port;
        ch_rate_p = @i8274$info.ch_a_in_rate_port;
        c$data.interrupting$unit = 0;
    END;
    ELSE
    DO;
        ch_p = @i8274$info.ch_b_data_port;
        ch_rate_p = @i8274$info.ch_b_in_rate_port;
        c$data.interrupting$unit = 1;
    END;

/*
 * Set up udata$p to point to the interrupting units data.
 * ( that is, add 1024 to the pointer for each unit )
 */

    udata$p = @cdata.udata;
    udata$p$o.offset = udata$p$o.offset +
                            SHL(DOUBLE(cdata.interrupting$unit),10);

    vector = (SHR(vector,2) AND 03H);
```

```
/*
 * Modify the vector so that Special Rx Condition interrupts
 * are handled in the Rx Char. Available case.
 */

    IF vector = 3 THEN
    DO;
        vector = 2;
        OUTPUT(ch.status_port) = ERROR_RESET;
        CALL delay(2);                    /* insure delay between outputs */
    END;

    IF vector = 2 THEN
    DO;
            /* Rx Char. available */

            char = INPUT(ch.data_port);

            /*
             *  If in auto baud rate search, check character for
             *  an identifiable baud rate
             */

            IF udata.in$rate <= AUTO$BAUD$SELECT THEN
            DO;
                char = char AND 07FH;
                IF (char = 55H) THEN
                    found$rate = 0;
                ELSE
                DO;
                    IF char = 66H THEN
                        found$rate = 1;
                    ELSE
                    DO;
                        IF char = 78H THEN
                            found$rate = 2;
                        ELSE
                        DO;
                            IF char = 0 THEN
                            DO;
                            /*
                             *  Go to next baud rate range and
                             *  condition terminal support to call setup
                             *  in about 150 ms.
                             */
                                udata.in$rate = udata.in$rate + 1;
                                IF udata.in$rate > AUTO$BAUD$SELECT THEN
                                    udata.in$rate = 1;
                                OUTPUT(ch.status_port) = WR1;
                                CALL delay(10); /* insure delay between outputs */
                                OUTPUT(ch.status_port) = WR1_NO_RX_INT;
                                CALL delay(10); /* insure delay between outputs */
                                cdata.interrupt$type = DELAY$INTERRUPT;
                                OUTPUT(ch.status_port) = WR3;
                                CALL delay(10); /* insure delay between outputs */
                                OUTPUT(ch.status_port) = WR3_RX_DISABLE;
                                CALL delay(10); /* insure delay between outputs */
                                /*  CALL TIME(10); */
                                OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;
                                RETURN char;
                            END;
                            ELSE
                            DO;
                                IF udata.in$rate <> 3 THEN
                                DO;
                                    cdata.interrupt$type = MORE$INTERRUPT;
                                    RETURN char;
                                END;
                                ELSE
```

```
                  DO;
                     udata.in$rate = 110;
                     OUTPUT(ch.status_port) = WR1;
                     CALL delay(10); /* insure delay between outputs */
                     OUTPUT(ch.status_port) = WR1_NO_RX_INT;
                     CALL delay(10); /* insure delay between outputs */
                     cdata.interrupt$type = DELAY$INTERRUPT;
                     OUTPUT(ch.status_port) = WR3;
                     CALL delay(10); /* insure delay between outputs */
                     OUTPUT(ch.status_port) = WR3_RX_DISABLE;
                     CALL delay(10); /* insure delay between outputs */
                     /*  CALL TIME(10); */
                     OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;
                     RETURN char;
                  END;
               END;
            END;
         END;
      END;

     /*
      *   Calculate recognized baud rate
      */

      udata.in$rate = SHR(19200, (udata.in$rate-1) * 3 + found$rate);
      OUTPUT(ch.status_port) = WR1;
      CALL delay(10); /* insure delay between outputs */
      OUTPUT(ch.status_port) = WR1_NO_RX_INT;
      CALL delay(10); /* insure delay between outputs */
      cdata.interrupt$type = DELAY$INTERRUPT;
      OUTPUT(ch.status_port) = WR3;
      CALL delay(10); /* insure delay between outputs */
      OUTPUT(ch.status_port) = WR3_RX_DISABLE;
      CALL delay(10); /* insure delay between outputs */
      /* CALL TIME(10); */
      OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;
      RETURN char;
   END;

/*
 *  check input parity mode & strip parity if desired
 */

   IF (udata.term$flags AND IN$PARITY$MASK) <> PASS$INPUT$PARITY$MODE
      THEN
   DO;
      IF (udata.term$flags AND IN$PARITY$MASK) =
         STRIP$INPUT$PARITY$MODE THEN
         char = char AND 07fh;
      ELSE
      DO;
         IF (udata.term$flags AND OUT$PAR$CHECK) <> 0 THEN
         DO;
            OUTPUT(ch.status_port) = RR1; /* point to RR1 */
            CALL delay(3);  /* insure delay between outputs */
            IF (input(ch.status_port) AND i8274$INPUT$ERROR) <> 0
               THEN
            DO;
               char = char OR 080H;
               OUTPUT(ch.status_port) = ERROR_RESET;
               CALL delay(10); /* insure delay between outputs */
               OUTPUT(ch.status_port) = WR3;
               CALL delay(10); /* insure delay between outputs */
               OUTPUT(ch.status_port) = WR3_INIT;
            END;
         END;
         ELSE
```

```
                    DO;
                        IF (udata.term$flags AND IN$PARITY$MASK) =
                           EVEN$INPUT$PARITY$MODE THEN
                        DO;
                            dummy = 0;
                            char = char OR dummy;
                            IF PARITY THEN
                                char = char AND 07FH;
                            ELSE
                                char = char OR 080H;
                        END;
                        ELSE
                        DO;
                            dummy = 0;
                            char = char OR dummy;
                            IF NOT PARITY THEN
                                char = char AND 07FH;
                            ELSE
                                char = char OR 080H;
                        END;
                    END;
                END;
            END;

            OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;

            cdata.interrupt$type = INPUT$INTERRUPT;

    END;
    ELSE
    DO;
        IF vector = 0 THEN
        DO; /* Tx Buffer empty */

            OUTPUT(ch.status_port) = RESET_TX_INT;
            CALL delay(5);                  /* insure delay between outputs */
            cdata.interrupt$type = OUTPUT$INTERRUPT;
            OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;

        END;
        ELSE
        DO; /* Ext/Status Change */

            OUTPUT(ch.status_port) = RESET_EXT_INT;
            cdata.interrupt$type = MORE$INTERRUPT;
            CALL delay(5);                  /* insure delay between outputs */
            OUTPUT(i8274$info.ch_a_status_port) = END_OF_INT;

        END; /* Ext/Status Change */
    END;

    RETURN char;

END i8274$check;
```

```
$subtitle('i8274$answer')

/*
 *  TITLE:  i8274$answer
 *
 *  CALLING SEQUENCE:
 *      CALL i8274$answer(udata$p);
 *
 *  INTERFACE VARIABLES:
 *      udata$p     POINTER to unit data
 *
 *  CALLS:
 *      none
 *
 *  ABSTRACT:
 *      Sends a mode word to the 8274 to place DTR active.
 *
 */

i8274$answer:   PROCEDURE(udata$p) REENTRANT PUBLIC;

DECLARE
    udata$p                              POINTER,
    udata$p$o   STRUCTURE(
                             offset      WORD,
                             base        SELECTOR) AT(@udata$p),
    cdata   BASED   udata$p$o.base  TS$CDATA,
    udata   BASED   udata$p         TS$UDATA;

DECLARE
    i8274$info$p                         POINTER,
    i8274$info  BASED   i8274$info$p     i8274$CONTROLLER$INFO;

DECLARE
    ch_p                                 POINTER,
    ch  BASED   ch_p    STRUCTURE (
                             data_port   WORD,
                             status_port WORD );

    i8274$info$p = cdata.dinfo$p;

    IF udata.unit$number = 0 THEN
        ch_p = @i8274$info.ch_a_data_port;
    ELSE
        ch_p = @i8274$info.ch_b_data_port;

    OUTPUT(ch.status_port) = WR5;
    CALL delay(10);                  /* insure delay between outputs */
    OUTPUT(ch.status_port) = WR5_DTR_ON;

END i8274$answer;
```

```
              $subtitle('i8274$hangup')

              /*
               *  TITLE:  i8274$hangup
               *
               *  CALLING SEQUENCE:
               *      CALL i8274$hangup(udata$p);
               *
               *  INTERFACE VARIABLES:
               *      udata$p     POINTER to unit data
               *
               *  CALLS:
               *      none
               *
               *  ABSTRACT:
               *      Sends a mode word to the 8274 to place DTR inactive.
               *
               */
402     1     i8274$hangup:   PROCEDURE(udata$p) REENTRANT PUBLIC;

403     2     DECLARE
                  udata$p                               POINTER,
                  udata$p$o   STRUCTURE(
                                          offset        WORD,
                                          base          SELECTOR) AT(@udata$p),
                  cdata   BASED   udata$p$o.base   TS$CDATA,
                  udata   BASED   udata$p          TS$UDATA;

404     2     DECLARE
                  i8274$info$p                          POINTER,
                  i8274$info  BASED   i8274$info$p      i8274$CONTROLLER$INFO;

405     2     DECLARE
                  ch_p                                  POINTER,
                  ch  BASED   ch_p    STRUCTURE (
                                          data_port     WORD,
                                          status_port   WORD );

406     2         i8274$info$p = cdata.dinfo$p;

407     2         IF udata.unit$number = 0 THEN
408     2             ch_p = @i8274$info.ch_a_data_port;

409     2         ELSE
                      ch_p = @i8274$info.ch_b_data_port;

410     2         OUTPUT(ch.status_port) = WR5;
411     2         CALL delay(10);             /* insure delay between outputs */
412     2         OUTPUT(ch.status_port) = WR5_DTR_OFF;

413     2     END i8274$hangup;
```

```
$subtitle('i8274$out')

/*
 *  TITLE:  i8274$out
 *
 *  CALLING SEQUENCE:
 *      CALL i8274$out(udata$p,char);
 *
 *  INTERFACE VARIABLES:
 *      udata$p     POINTER to unit data
 *      char        BYTE to OUTPUT
 *
 *  CALLS:
 *      none
 *
 *  ABSTRACT:
 *      OUTPUTs a char to selected channel of the 8274.
 *      Marking or spacing parity is handled here if enabled,
 *      and the char is sent out.
 *
 */
i8274$out: PROCEDURE(udata$p,char) PUBLIC REENTRANT;

DECLARE
    udata$p                             POINTER,
    udata$p$o   STRUCTURE(
                            offset      WORD,
                            base        SELECTOR) AT(@udata$p),
    cdata   BASED   udata$p$o.base  TS$CDATA,
    udata   BASED   udata$p         TS$UDATA;

DECLARE
    i8274$info$p                        POINTER,
    i8274$info  BASED   i8274$info$p    i8274$CONTROLLER$INFO;

DECLARE
    ch_p                                POINTER,
    ch  BASED   ch_p    STRUCTURE (
                            data_port   WORD,
                            status_port WORD );

DECLARE
    char                                BYTE,
    mode                                WORD;

    i8274$info$p = cdata.dinfo$p;

    IF udata.unit$number = 0 THEN
        ch_p = @i8274$info.ch_a_data_port;

    ELSE
        ch_p = @i8274$info.ch_b_data_port;

    mode = udata.term$flags AND OUT$PARITY$MASK;
    IF mode <= MARK$OUTPUT$PARITY$MODE THEN
    DO;
        IF mode = MARK$OUTPUT$PARITY$MODE THEN
            char = char OR 80H;
        ELSE
            char = char AND 07FH;
    END;

    OUTPUT(ch.data_port) = char;


END i8274$out;
```

```
$subtitle('i8274$finish')

/*
 *  TITLE:  i8274$finish
 *
 *  CALLING SEQUENCE:
 *      CALL i8274$finish(cdata$p);
 *
 *  INTERFACE VARIABLES:
 *      cdata$p - pointer to controller data.
 *
 *  CALLS:
 *      none
 *
 *  ABSTRACT:
 *      Procedure disables TX, RX and interrupts.
 *
 */
i8274$finish: PROCEDURE (cdata$p) PUBLIC REENTRANT;

DECLARE
    cdata$p                              POINTER,
    cdata   BASED   cdata$p              TS$CDATA;

DECLARE
    i8274$info$p                         POINTER,
    i8274$info  BASED   i8274$info$p     i8274$CONTROLLER$INFO;

DECLARE
    port                                 WORD;


/*
 *     Get the configuration info
 */
   i8274$info$p = cdata.cinfo$p;

/*
 *  Disable the 8274 TX, RX, and interrupts.
 */

    port = i8274$info.ch_b_status_port;

    OUTPUT(port) = WR5;               /* point to WR5 */
    CALL delay(10);                   /* insure delay between outputs */
    OUTPUT(port) = WR5_TX_DISABLE;    /* disable Tx
    CALL delay(10);                   /* insure delay between outputs */

    OUTPUT(port) = WR3;               /* point to WR3 */
    CALL delay(10);                   /* insure delay between outputs */
    OUTPUT(port) = WR3_RX_DISABLE;    /* disable Rx
    CALL delay(10);                   /* insure delay between outputs */

    OUTPUT(port) = WR1;               /* point to WR1 */
    CALL delay(10);                   /* insure delay between outputs */
    OUTPUT(port) = WR1_NO_INT;        /* disable interrupts
    CALL delay(10);                   /* insure delay between outputs */

    port = i8274$info.ch_a_status_port;

    OUTPUT(port) = WR5;               /* point to WR5 */
    CALL delay(10);                   /* insure delay between outputs */
    OUTPUT(port) = WR5_TX_DISABLE;    /* disable Tx
    CALL delay(10);                   /* insure delay between outputs */

    OUTPUT(port) = WR3;               /* point to WR3 */
    CALL delay(10);                   /* insure delay between outputs */
    OUTPUT(port) = WR3_RX_DISABLE;    /* disable Rx
    CALL delay(10);                   /* insure delay between outputs */

    OUTPUT(port) = WR1;               /* point to WR1 */
    CALL delay(10);                   /* insure delay between outputs */
    OUTPUT(port) = WR1_NO_INT;        /* disable interrupts
    CALL delay(10);                   /* insure delay between outputs */

END i8274$finish;

END x8274;
```

```
   MODULE INFORMATION:

        CODE AREA SIZE     = 0943H   2371D
        CONSTANT AREA SIZE = 0000H      0D
        VARIABLE AREA SIZE = 0000H      0D
        MAXIMUM STACK SIZE = 0030H     48D
        1394 LINES READ
        0 PROGRAM WARNINGS
        0 PROGRAM ERRORS

   DICTIONARY SUMMARY:

        96KB MEMORY AVAILABLE
        22KB MEMORY USED   (22%)
        0KB DISK SPACE USED

   END OF PL/M-86 COMPILATION
/*
 *    x8255.lit
 *
 *  8255 is programmed as follows:
 *
 *    Group A:  Mode 1
 *    Group B:  Mode 0
 *
 *    Port A and Lower Port C: OUTPUT
 *    Port B and Upper Port C: INPUT
 *
 *  Port C definition (bit 0 is LSB;  bit 7 is MSB):
 *
 *      Bit  0   -   Character strobe to the printer
 *           1   -   not used
 *           2   -   not used
 *           3   -   Character acknowledge from the printer is complete
 *           4   -   Printer ready
 *           5   -   Paper out
 *           6   -   Printer interrupt enable
 *           7   -   Character acknowledge from the printer
 */
DECLARE
   MODE$WORD          LITERALLY   '0AAH',
   CHAR$ACK$COMPLETE  LITERALLY   '08H',
   PRINTER$READY      LITERALLY   '10H',
   PAPER$OUT          LITERALLY   '20H',
   CHAR$ACK           LITERALLY   '80H',
   INT$ENABLE         LITERALLY   '0DH',
   INT$DISABLE        LITERALLY   '0CH',
   STROBE$ON          LITERALLY   '01H',
   STROBE$OFF         LITERALLY   '00H';


    /*
     * xprntr.lit
     *
     * Common device driver information
     *
     * level:                 Interrupt level
     * priority:              Priority of interrupt task
     * stack$size:            Stack size for interrupt task
     * data$size:             Device local data size
     * num$units:             Number of units on device
     * device$init:           Init device procedure
     * device$finish:         Finished with device procedure
     * device$start:          Start device procedure
     * device$stop:           Stop device procedure
     * device$interrupt:      Device interrupt procedure
     */

DECLARE COMMON$DEV$INFO LITERALLY
    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';
```

```
DECLARE i8255$INFO LITERALLY
    A$port              WORD,
    B$port              WORD,
    C$port              WORD,
    Control$port        WORD';



DECLARE
    PRINTER$DEVICE$INFO LITERALLY 'STRUCTURE(
        COMMON$DEV$INFO,
        i8255$INFO,
        tab$control   WORD)';


$save nolist
/*
 * x206dv.lit
 *      Defines literals for 206 driver
 *
 */

    /*
     * The iopb fields (first 9 bytes) must be first!!
     * They are used later and the other procedures
     * do not know of status or restore.
     *
     * Note that each spindle has up to 4 platters, and each 206 can support
     * up to 4 spindles.  Thus, there are 4 statuses:  one for each spindle.
     *
     * Restore is used to indicate that there is a restore in progress.
     * It is set when a restore is started after a request returns an
     * error which requires a restore to reset the drive.  A new request
     * is not started when there is a restore in progress, instead the
     * interrupt routine starts the request and resets restore when
     * the restore finishes.
     */

    DECLARE
        IO$PARM$BLOCK$206   LITERALLY 'STRUCTURE(
            inter               BYTE,
            instr               BYTE,
            r$count             BYTE,
            cyl$add             BYTE,
            rec$add             BYTE,
            buff$p              POINTER,
            status(4)           BYTE,
            restore             BYTE,
            format$table(72)    BYTE)';

    /*
     *  defines masks
     */

    DECLARE
        inter$on$mask       LITERALLY '008H',   /* bit 4 := 16-bit data */
        inter$off$mask      LITERALLY '018H',
        FORMAT$TRACK$ON     LITERALLY '040H',
        i206$TRACK$MAX      LITERALLY '800',    /* 400 tracks * 2 surfaces */
        i206$SECTOR$MAX     LITERALLY '36',
        command$busy        LITERALLY '080H';

    /*
     *  defines op-codes
     */

    DECLARE
        no$op               LITERALLY '00H',
        seek$op             LITERALLY '01H',
        format$op           LITERALLY '02H',
        restore$op          LITERALLY '03H',
        read$op             LITERALLY '04H',
        verify$op           LITERALLY '05H',
        write$op            LITERALLY '06H';
    /*
     *  defines ports
     */

    declare
        sub$system$port     LITERALLY 'base',
        result$type$port    LITERALLY 'base + 1',
        controller$stat     LITERALLY 'base + 2',
        result$byte$port    LITERALLY 'base + 3',
        inter$stat$port     LITERALLY 'base + 4',
        disk$config$port    LITERALLY 'base + 7',
        lo$seg$port         LITERALLY 'base',
        hi$seg$port         LITERALLY 'base',
        lo$off$port         LITERALLY 'base + 1',
        hi$off$port         LITERALLY 'base + 2',
        start$diagnostic    LITERALLY 'base + 5',
        reset$port          LITERALLY 'dinfo.base + 7';

$restore


$save nolist

/*
 *  x206in.lit
 *
 *  206 Driver info
 *  Adds to the device$info and unit$info structures,
 *  using common device support and random access
 *  device support.
 *
 */

/*
 *  Per device information
 */

    DECLARE
        I206$DEVICE$INFO LITERALLY 'STRUCTURE(
            RADEV$DEVICE$INFO,
            base        WORD)';

/*
 *  Per unit information
 */

    DECLARE
        I206$UNIT$INFO LITERALLY 'STRUCTURE(RAD$UNIT$INFO)';
$restore


$save nolist
    /*
     * x206dc.ext
     */

    send$206$iopb: PROCEDURE (base, iopb$p) BOOLEAN EXTERNAL;
        DECLARE
            base        WORD,
            iopb$p      POINTER;
    END send$206$iopb;

$restore
```

```
$save nolist

/*
 *  x206dp.ext
 */

    io$206: PROCEDURE (base, iors$p, duib$p, iopb$p) EXTERNAL;
        DECLARE
            base        WORD,
            iors$p      POINTER,
            duib$p      POINTER,
            iopb$p      POINTER;
    END io$206;

$restore


$save nolist

/*
 *  x206fm.ext
 */

    format$206: PROCEDURE (base, iors$p, duib$p, iopb$p) EXTERNAL;
        DECLARE
            base        WORD,
            iors$p      POINTER,
            duib$p      POINTER,
            iopb$p      POINTER;
    END format$206;

$restore


$SAVE NOLIST

/*
 *  nsleep.ext
 */

rq$sleep:  PROCEDURE( time$limit,
                      except$ptr ) EXTERNAL;

DECLARE time$limit    WORD,
        except$ptr    POINTER;

END rq$sleep;

$RESTORE


$save nolist
    /*
     * xcomon.lit
     *      Oft-used literals.
     *
     */

    DECLARE
        BOOLEAN         LITERALLY 'BYTE',
        TRUE            LITERALLY '0FFH',
        FALSE           LITERALLY '000H',
        FOREVER         LITERALLY 'WHILE TRUE',
        PTR$OVERLAY     LITERALLY 'STRUCTURE(offset WORD, base TOKEN)',
        P$OVERLAY       LITERALLY 'STRUCTURE(offset WORD, base WORD)',
        STRING          LITERALLY 'STRUCTURE(length BYTE, char(1) BYTE)',
/*
        DWORD           LITERALLY 'POINTER',
 */
        NO$TIME$LIMIT   LITERALLY '0FFFFH',
        BYTE$MAX        LITERALLY '0FFH',
        WORD$MAX        LITERALLY '0FFFFH',
        FIFO$Q          LITERALLY '000H',       /* select FIFO queueing */
        PRIO$Q          LITERALLY '001H';       /* select PRIO queueing */
$restore


$SAVE NOLIST

/*
 *  xdelay.ext
 */

/*
 * External Declaration for Delay Procedure.
 */

delay:  PROCEDURE(units) EXTERNAL;

    DECLARE
        units                           BYTE;

END delay;

$restore


$save nolist
/*
 * xdrinf.lit
 *      Driver information for common and random access devices.
 *
 */

    /*
     * Random-access driver information
     *
     * level:               Interrupt level
     * priority:            Priority of interrupt task
     * stack$size:          Stack size for interrupt task
     * data$size:           Device local data size
     * num$units:           Number of units on device
     * device$init:         Init device procedure
     * device$finish:       Finished with device procedure
     * device$start:        Start device procedure
     * device$stop:         Stop device procedure
     * device$interrupt:    Device interrupt procedure
     */

    DECLARE
        RADEV$DEVICE$INFO LITERALLY
            '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';
    /*
     * Unit info for radev
     *
     * track$size:      Size in bytes of track.  Used for calculating
     *                  track/sector.  Requests to device will not cross
     *                  track boundaries.
     * max$retry:       Number of times to retry on a soft IO error.
     */

    DECLARE
        RAD$UNIT$INFO LITERALLY
           'track$size          WORD,
            max$retry           WORD,
            cylinder$size       WORD';
$restore


$save nolist
    /*
     * xduib.lit
     *      Device-Unit Information Block definition.
     *
     */

    /*
     * name:            ASCII name of dev-unit, null padded
     * file$driver:     bit(i) ==> file-driver (i+1) is ok for this device.
     *                  See idevmg.plm
     * functs:          from EPS, bit i ==> function(i) supported by the driver.
     * flags:           For 215 only.  See EPS.
     *                  functions are F$FORMAT, F$READ, etc.
     * dev$gran:        device granularity in bytes.
     * dev$size:        size (in bytes) of device-unit
     * device:          device number/device code
     * unit:            device specific number of controller sub-unit (i.e.,
     *                  for a 204, could be 0,1 to indicate different drives)
     * dev$unit:        unique number identifying a device/unit pair for device
     *                  allocation purposes
     * init$io:         driver procedure for initializing driver
     * finish$io:       driver procedure for turning off/deallocating driver
     * queue$io:        driver procedure for queueing I/O requests
     * cancel$io:       driver procedure for cancelling I/O requests
     * device$info$p:   device specific information pointer.
     * unit$info$p:     unit specific information pointer.
     * update$timeout:  time (ticks) before update on this unit
     * num$buffers:     number of deblocking/buffering buffers for this unit
     * priority:        service task priority.
     * fixed$update:    boolean to indicate use of wall clock updates.
     * max$buffers:     maximum no. of buffers for device (used by EIOS)
     * fill:            filler byte
     */

    DECLARE
        DUIB$PART$ONE LITERALLY
           'name(DEV$NAME$LEN)  BYTE,
            file$driver     WORD,
            functs          BYTE,
            flags           BYTE,
            dev$gran        WORD,
            dev$size        DWORD,
            device          BYTE,
            unit            BYTE,
            dev$unit        WORD',
        DUIB$PART$TWO LITERALLY
           '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,
            fixed$update   BYTE,
            max$buffers    BYTE,
            fill           BYTE',
        DEV$UNIT$INFO$BLOCK LITERALLY 'STRUCTURE(
            DUIB$PART$ONE,
            DUIB$PART$TWO)';

    DECLARE
        VF$AUTO     LITERALLY '1',
        VF$DENSITY  LITERALLY '2',
        VF$SIDES    LITERALLY '4',
        VF$MINI     LITERALLY '8';
$restore


$save nolist
    /*
     * xexcep.lit
     *      I/O System Exception Code Mnemonics.
     */

$include(:f1:xnerro.lit)

    /*
     * IOS Synchronous Avoidable exception codes.
     */

    DECLARE
        E$NOUSER            LITERALLY '08021H',    /* Job has no Default User Object */
        E$NOPREFIX          LITERALLY '08022H';    /* Job has no Default Prefix Object */

    /*
     * IOS Asynchronous exception codes.
     */

    DECLARE
        E$FEXIST            LITERALLY '00020H',    /* File Exists */
        E$FNEXIST           LITERALLY '00021H',    /* Non-existant File */
        E$DEVFD             LITERALLY '00022H',    /* Device & File Driver Incompatable */
        E$SUPPORT           LITERALLY '00023H',    /* Un-supported Request */
        E$EMPTY$ENTRY       LITERALLY '00024H',    /* Empty Directory Entry */
        E$DIR$END           LITERALLY '00025H',    /* End of Directory */
        E$FACCESS           LITERALLY '00026H',    /* Access to File Not Granted */
        E$FTYPE             LITERALLY '00027H',    /* Bad File Type */
        E$SHARE             LITERALLY '00028H',    /* Improper File Sharing Requested */
        E$SPACE             LITERALLY '00029H',    /* No Space Left */
        E$IDDR              LITERALLY '0002AH',    /* Illegal Device Driver Request */
        E$IO                LITERALLY '0002BH',    /* I/O Error */
        E$FLUSHING          LITERALLY '0002CH',    /* Connection is flushing requests */
        E$ILLVOL            LITERALLY '0002DH',    /* Illegal Volume */
        E$DEV$OFF$LINE      LITERALLY '0002EH',    /* Device Was Off Line */
        E$IFDR              LITERALLY '0002FH';    /* Illegal File Driver Request */

    /*
     * E$IO expanded with unitstatus codes
     */
    DECLARE
        E$IO$UNCLASS        LITERALLY '00050H',     /* Unclassified */
        E$IO$SOFT           LITERALLY '00051H',     /* Soft error */
        E$IO$HARD           LITERALLY '00052H',     /* Hard error */
        E$IO$OPRINT         LITERALLY '00053H',     /* Operator intervention required */
        E$IO$WRPROT         LITERALLY '00054H',     /* Write protected */
        E$IO$NO$DATA        LITERALLY '00055H',     /* No further data */
        E$IO$MODE           LITERALLY '00056H';     /* Mode violation */

$restore


$save nolist

/*
 *  xioexc.lit
 */

/*
 *  IO exception codes
 */

    DECLARE
        IO$UNCLASS      LITERALLY '0',
        IO$SOFT         LITERALLY '1',
        IO$HARD         LITERALLY '2',
        IO$OPRINT       LITERALLY '3',
        IO$WRPROT       LITERALLY '4',
        IO$NO$DATA      LITERALLY '5',
        IO$MODE         LITERALLY '6';
$restore


$save nolist
    /*
     * xiofct.lit
     *
     *  IO function codes
     *
     */

    DECLARE
        F$READ          LITERALLY '0',
        F$WRITE         LITERALLY '1',
        F$SEEK          LITERALLY '2',
        F$SPECIAL       LITERALLY '3',
        F$ATTACH$DEV    LITERALLY '4',
        F$DETACH$DEV    LITERALLY '5',
        F$OPEN          LITERALLY '6',
        F$CLOSE         LITERALLY '7',
        F$GETCS         LITERALLY '8',
        F$GETFS         LITERALLY '9',
        F$GETEXT        LITERALLY '10',
        F$SETEXT        LITERALLY '11',
        F$NULL$CH$ACCESS LITERALLY '12',
        F$NULL$DELETE   LITERALLY '13',
        F$RENAME        LITERALLY '14',
        F$GET$PATH$COMP LITERALLY '15',
        F$GET$DIR$ENTRY LITERALLY '16',
        F$TRUNC         LITERALLY '17',
        F$DETACH        LITERALLY '18',
        F$NUM$FUNCT     LITERALLY '19';
```

```
/*
 * Function codes for internal use only.
 * The rq$common$attach and common$io$task use F$ATTACH$THRU.
 * The req$update and common$io$task use F$UPDATE.
 */

    DECLARE
        F$ATTACH$THRU   LITERALLY '19',
        F$UPDATE        LITERALLY '20';

$restore


$save nolist
    /*
     * xiotyp.lit
     *      RMX/86 I/O System "type" literals.
     *
     */

    DECLARE
        CONNECTION  LITERALLY 'TOKEN',
        USER        LITERALLY 'TOKEN',
        BLOCK$NUM   LITERALLY '(3) BYTE';
$restore


$save nolist
    /*
     * xiors.lit
     *      I/O Request/Result Segment
     *
     */

    DECLARE
        IORS$PART$ONE LITERALLY
           'status      WORD,
            unit$status WORD,
            actual      WORD,
            actual$fill WORD,
            device      WORD,
            unit        BYTE,
            funct       BYTE,
            subfunct    WORD,
            dev$loc     DWORD,
            buff$p      POINTER',

        IORS$PART$TWO LITERALLY
           'count       WORD,
            count$fill  WORD,
            aux$p       POINTER,
            link$for    POINTER,
            link$back   POINTER,
            resp$mbox   MAILBOX,
            done        BOOLEAN,
            iors$fill   BYTE,
            cancel$id   TOKEN,
            conn$t      TOKEN',

        IO$REQ$RES$SEG  LITERALLY 'STRUCTURE(
            IORS$PART$ONE,
            IORS$PART$TWO)';

    /*
     * Define number of actual bytes of data (i.e., before links)
     */

    DECLARE
        IORS$DATA$SIZE  LITERALLY '30';
```

```
$include(:f1:xiofct.lit)
$restore


$save nolist

/*
 *  xnotif.ext
 *
 *  External for notify support procedure
 *  Called by random access supported drivers
 */

notify: PROCEDURE(unit, ddata$p) EXTERNAL;
    DECLARE
        unit         BYTE,
        ddata$p      POINTER;
END notify;

$restore


$save nolist

/*
 *  xnerro.lit
 */

    DECLARE
        E$OK                 LITERALLY '00000H',
        E$TIME               LITERALLY '00001H',
        E$MEM                LITERALLY '00002H',
        E$BUSY               LITERALLY '00003H',
        E$LIMIT              LITERALLY '00004H',
        E$CONTEXT            LITERALLY '00005H',
        E$EXIST              LITERALLY '00006H',
        E$STATE              LITERALLY '00007H',
        E$NOT$CONFIGURED     LITERALLY '00008H';

    DECLARE
        E$ZERO$DIVIDE        LITERALLY '08000H',
        E$OVERFLOW           LITERALLY '08001H',
        E$TYPE               LITERALLY '08002H',
        E$BOUNDS             LITERALLY '08003H',
        E$PARAM              LITERALLY '08004H',
        E$BAD$CALL           LITERALLY '08005H';

$restore

$save nolist
    /*
     * xnutyp.lit
     *      RMX/86 Nucleus "type" literals.
     *
     */

    DECLARE
        TOKEN        LITERALLY 'SELECTOR',
        SEGMENT      LITERALLY 'TOKEN',
        TASK         LITERALLY 'TOKEN',
        REGION       LITERALLY 'TOKEN',
        SEMAPHORE    LITERALLY 'TOKEN',
        MAILBOX      LITERALLY 'TOKEN',
        JOB          LITERALLY 'TOKEN',
        EXTENSION    LITERALLY 'TOKEN';

    DECLARE
        T$MAILBOX    LITERALLY '03H',
        T$SEGMENT    LITERALLY '06H';
$restore
```

```
$save nolist
    /*
     * xparam.lit
     *      I/O System parameter literals.
     *
     */

    DECLARE
        DEV$NAME$LEN    LITERALLY '14',     /* device name is 14 bytes */
        PATH$COMP$LEN   LITERALLY '14',     /* path component size */
        UP$COMP         LITERALLY '''^''',  /* "up" component character */
        PATH$SEP        LITERALLY '''/''',  /* path component seperator character */
        DEF$PREFIX$CHAR LITERALLY '''$''';  /* default-prefix character */

    DECLARE
        ATT$DEV$TASK$STACK$SIZE         LITERALLY '512',
        CONN$JOB$DELETE$TASK$STACK$SIZE LITERALLY '512',
        TIMER$TASK$STACK$SIZE           LITERALLY '512',
        COMMON$DRIVER$STACK$SIZE        LITERALLY '512';

    DECLARE
        IOS$OS$EXTENSION    LITERALLY '192';        /* OS extension vector */

    DECLARE
        XFACE$Q$LEN         LITERALLY '(5*2)',      /* xface mbox queue length = 5*4 */
        CONN$DEL$Q$LEN      LITERALLY '(5*2)';      /* conn job-del mbox queue length = 5*4 */
$restore


$save nolist

/*
 *  xprerr.lit
 */

/*
 *  error codes
 */
DECLARE
   E$OK        LITERALLY '0000H',
   E$IDDR      LITERALLY '002AH';
$restore


$save nolist

/*
 *  xtrsec.lit
 */

DECLARE TRACK$SECTOR$STRUCT LITERALLY 'STRUCTURE(
    sector  WORD,
    track   WORD)';
$restore


$save nolist

/*
 *  xtssow.ext
 */

xts$set$output$waiting: PROCEDURE(udata$p) EXTERNAL;
DECLARE
    udata$p             POINTER;
END xts$set$output$waiting;

$restore
```

```
$SAVE NOLIST

/*
 *  xtstim.ext
 */

/*
 * External Declaration
 * for timer support procedure.
 */

set$baud$rate$count: PROCEDURE(command_port, counter_port, timer_type,
                        counter_number, rate_count) EXTERNAL;

DECLARE
    (command_port, counter_port, rate_count)    WORD,
    (timer_type, counter_number)                BYTE;

END set$baud$rate$count;

$RESTORE


$save nolist
    /*
     * xradsf.lit
     *     Random-Access driver Special-Function Mnemonics.
     *
     */

    DECLARE
        FS$FORMAT$TRACK LITERALLY '0';      /* format a track */

    /*
     *  Format info structure to format one track on
     *  a disk(hard or floppy)
     *  used by 204 & 206 drivers
     *
     */

    DECLARE
        FORMAT$INFO$STRUCT  LITERALLY 'STRUCTURE(
            track$num           WORD,
            track$interleave    WORD,
            track$skew          WORD,
            fill$char           BYTE)';

    /*
     *  Device label special function.  Asks driver to supply
     *  device information for named file label.
     *
     */

    DECLARE
        FS$DEVICE$LABEL     LITERALLY '3';

    /*
     *  Special tape functions.
     *
     */

    DECLARE
        FS$REWIND           LITERALLY '7',
        FS$READ$FILE$MARK   LITERALLY '8',
        FS$WRITE$FILE$MARK  LITERALLY '9',
        FS$RETENSION        LITERALLY '10';

$restore
```

***

INDEX

Primary references are underscored.

assembly language iii, 8-1
attach device requests 4-1

baud rate 7-16, 7-19
BEGIN$LONG$TERM$OP procedure 5-8
buffered devices 7-17, 7-25
buffers 2-6, 2-7

CANCEL$IO procedure 2-5, 3-3, 6-4, A-6
cancel requests 4-2
close requests 4-2
common device driver 1-3, 5-1
  device information table 3-8
  example B-2
  support routines A-1, B-55
common device 3-1
communication levels 1-1
configuration 2-1, 8-1
connection 2-16
creating DUIBs 2-8
custom device drivers 1-3, 6-1
custom devices 3-2
cylinder 3-11

data storage area 3-9, 3-13
data structures 3-7, 3-12
DEFAULT$FINISH procedure 5-3
DEFAULT$INIT procedure 5-2
DEFAULT$STOP procedure 5-5
detach device requests 4-2
device
  buffered 7-17, 7-25
  granularity 2-4
  interfaces 2-13
  number 1-2, 2-5, 2-11
device data storage area 3-9, 3-13, A-3, A-9
device driver
  interfaces 2-1
  sample INCLUDE files B-55
  type 1-3
DEVICE$FINISH procedure 3-10, 5-2
Device Information Table 2-5, 3-8, 7-3, 7-27
DEVICE$INIT procedure 3-10, 5-2
DEVICE$INTERRUPT procedure 3-10, 5-5
DEVICE$START procedure 3-10, 5-3

DEVICE$STOP procedure 3-10, 5-4
device-unit information block (DUIB) 2-2
    creation 2-8
    structure 2-2
    use of 2-7, 4-3
device-unit
    name 2-3
    number 1-2, 2-5
doubly linked list 6-5
driver configuration 8-1
DQ$ATTACH system call 2-3, 3-4
DQ$CREATE system call 2-3, 3-4
DUIB, see: device-unit information block

END$LONG$TERM$OP procedure 5-9
examples of device drivers B-1
    device driver INCLUDE files B-55
    disk controller driver B-8
    printer driver B-2
    terminal driver B-29

file connection 2-16
file drivers 1-2, 2-3
FINISH$IO procedure 2-5, 3-2, 3-10, 6-2, A-3
fixed updating 2-6
FORMAT command 5-11
functions 2-3, 2-11

GET$IORS procedure 5-10
granularity 2-4, 2-7

I206DS.P86 disk-controller driver source file B-8
INIT$IO procedure 2-5, 3-2, A-1, A-9
Intel-supplied routines 5-1
Interactive Configuration Utility (ICU) 8-1
interfaces to the device driver 2-1
interrupt
    handlers and tasks 3-3
    level 3-8, 7-5
    task A-3
    task priority 3-9
    type 7-13, 7-22
INTERRUPT$TASK procedure 3-3, A-9
I/O functions 2-3
I/O request/result segment (IORS) 1-3, 2-9, A-5, A-7, A-11
    structure 2-9
    use of 4-3
I/O requests 1-3, 4-1
I/O System interfaces 2-1
I/O System responses 4-1
I/O System-supplied routines 5-1
IPRNTR.P86 printer driver source file B-2

levels of communication 1-1
linked list 6-5

long-term operations 5-8

modem 7-8

name of device-unit 2-2
notify procedure 2-14, 5-6
numbering of devices 1-2

open requests 4-2

parity 7-7
PL/M-86 iii, 5-11, 8-1
portable device drivers 3-14
priority 3-9

QUEUE$IO procedure 2-5, 3-3, 6-3, A-5

RAD$ procedure-name prefix (iRMX 88 systems only) 3-2, A-1
random access device drivers 1-3, 5-1
random access devices 3-1
random access driver example B-8
read requests 4-2
request queue 6-5
requests 1-3, 4-1
requirements for using the common device driver 3-1
retry limit 3-11
RQ$A$PHYSICAL$ATTACH$DEVICE system call 2-3, 3-4, 3-5, 6-4, A-1
RQ$ELVL system call A-9
RQ$FORMAT system call 5-11
RQ$SET$INTERRUPT system call A-9

SEEK$COMPLETE procedure 3-11, 5-7
seek requests 4-2
set output waiting (XTS$SET$OUTPUT$WAITING) procedure 7-18, 7-24
signal character 2-15
source files, device drivers B-1
special requests 4-2
stack size 3-9
support (INCUDE) files B-55

tape drives 2-14, 5-8
  rewinding of 5-8
terminal
  attributes 2-15
  baud rate 7-16, 7-19
  Device Information Table 2-5, 3-8, 7-3, 7-27
  devices 3-3
  driver example B-29
  drivers 7-1
  flags 7-8, 7-14
  modem 7-8
  parity 7-7
terminal answer (TERM$ANSWER) procedure 7-17, 7-20, 7-27
terminal check (TERM$CHECK) procedure 7-17, 7-22, 7-27

terminal controller data 7-14, 7-27
terminal finish (TERM$FINISH) procedure 7-17, 7-19, 7-27
terminal hangup (TERM$HANGUP) procedure 7-17, 7-21, 7-27
terminal initialization (TERM$INIT) procedure 7-17, 7-18, 7-27
terminal output (TERM$OUT) procedure 7-17, 7-24, 7-27
terminal setup (TERM$SETUP) procedure 7-17, 7-19, 7-27
Terminal Support Code 7-11
terminal unit data 7-4, 7-14, 7-27
track size 3-11
types of device drivers 1-3

Unit Information Table 2-5, 3-10, 7-6
unit number 1-2, 2-5, 2-11
unit status codes 2-10
updating output to a device 2-6
using DUIBs 2-7

volume granularity 2-7

write requests 4-2

X8274.P86 terminal driver source file B-29
XTS$SET$OUTPUT$WAITING procedure 7-24

***