digital

VAX/VMS

I/O User's Guide

Order No. AA-D028B-TE

VAX11

March 1980

This document contains the information necessary to interface directly with the I/O device drivers supplied as part of the VAX/VMS operating system. Several examples of programming techniques are included. This document does not contain information on I/O operations using VAX-11 Record Management Services.

VAX/VMS
I/O User's Guide

Order No. AA-D028B-TE

SUPERSESSION/UPDATE INFORMATION:	This revised document supersedes the VAX/VMS I/O User's Guide (Order No. AA-D028A-TE)
OPERATING SYSTEM AND VERSION:	VAX/VMS V02
SOFTWARE VERSION:	VAX/VMS V02

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, August 1978
Revised, March 1980

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL
DEC
PDP
DECUS
UNIBUS
COMPUTER LABS
COMTEX
DDT
DECCOMM
ASSIST-11
VAX
DECnet
DATATRIEVE
DECsystem-10
DECtape
DIBOL
EDUSYSTEM
FLIP CHIP
FOCAL
INDAC
LAB-8
DECSYSTEM-20
RTS-8
VMS
IAS
TRAX
MASSBUS
OMNIBUS
OS/8
PHA
RSTS
RSX
TYPESET-8
TYPESET-11
TMS-11
ITPS-10
SBI
PDT

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

FIGURES

CONTENTS

FIGURES (Cont.)

CONTENTS

FIGURES (Cont.)

TABLES

TABLES (Cont.)

CONTENTS

TABLES (Cont.)

PREFACE

MANUAL OBJECTIVES

This manual provides users of the VAX/VMS operating system with the information necessary to interface directly with the I/O device drivers supplied as part of the operating system. It is not the objective of this manual to provide the reader with information on all aspects of VAX/VMS input/output (I/O) operations.

INTENDED AUDIENCE

This manual is intended for system programmers who want to take advantage of the time and/or space savings that result from direct use of the I/O devices. Users of VAX/VMS who do not require such detailed knowledge of I/O drivers can use the device-independent services described in the VAX-11 Record Management Services Reference Manual Readers are expected to have some experience with either VAX-11 FORTRAN or VAX-11 MACRO assembly language.

STRUCTURE OF THIS DOCUMENT

This manual is organized into thirteen chapters and one appendix, as follows:

- Chapter 1 contains introductory information. It provides overviews of VAX/VMS I/O operations; I/O system services; and I/O quotas, privileges, and protection. This chapter describes I/O function encoding and how to make I/O requests. It also describes how to obtain information on the different devices.

- Chapters 2 through 8 and 10 through 12 describe the use of all the I/O device drivers supported by VAX/VMS:

  - Chapter 2 deals with the terminal driver

  - Chapter 3 deals with disk drivers

  - Chapter 4 deals with magnetic tape drivers

  - Chapter 5 deals with the line printer driver

  - Chapter 6 deals with the card reader driver

  - Chapter 7 deals with the mailbox driver

- Chapter 8 deals with the DMC11 driver
- Chapter 10 deals with the LPA11-K driver
- Chapter 11 deals with the DR-32 driver
- Chapter 12 deals with the DUP11 driver

- Chapter 9 describes the Queue I/O (QIO) interface to file system ancillary control processes (ACPs).
- The appendix summarizes the QIO function codes, arguments, and function modifiers used by the different device drivers.

ASSOCIATED DOCUMENTS

The following documents may also be useful:

- *VAX/-11 Information Directory and Index* - contains a complete list of all VAX-11 documents
- *VAX/VMS System Services Reference Manual*
- *VAX-11 Linker Reference Manual*
- *VAX-11 Software Handbook*
- *PDP-11 Peripherals Handbook*
- *VAX-11 FORTRAN User's Guide*
- *VAX-11 MACRO User's Guide*
- *VAX-11 Record Management Services Reference Manual*
- *LPA11-K Laboratory Peripheral Accelerator User's Guide*
- *DECnet-VAX User's Guide*
- *VAX/VMS 2780/3780 Protocol Emulator User's Guide*

CONVENTIONS USED IN THIS MANUAL

The following conventions are used in this manual.

Convention	Meaning
[]	Brackets in QIO requests enclose optional arguments. For example: IO$_CREATE P1,[P2],[P3],[P4],[P5]
...	Horizontal ellipses indicate that characters or QIO arguments not pertinent to the example have been omitted. For example: (that is, 8, 16, 24,...).

Convention **Meaning**

.
.
.

Vertical ellipses in coding examples indicate that lines of code not pertinent to the example are omitted. For example:

```
     TTCHAN:  .BLKW 1

                    .
                    .
                    .

              $ASSIGN_S DEVNAM=TTNAME,CHAN=TTCHAN
```

-

Hyphens in coding examples indicate that additional arguments to the QIO request are provided on the following line(s). For example:

```
$QIO_S  FUNC=#IO$_WRITEPBLK,-        ;FUNCTION IS
        -                            ;WRITE PHYSICAL
        CHAN=W^TTCHAN1,-             ;TO TTCHAN 1
        EFN=#1,-                     ;EVENT FLAG 1
        P1=W^ASTMSG,-                ;P1 = BUFFER
        P2=#ASTMSGSIZE               ;P2 = BUFFER SIZE
```

<>

Angle brackets enclose keys on the terminal keyboard. For example:

```
     <0> <20-2F>...<40-7E>
```

numbers

Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes -- binary, octal, or hexadecimal -- are explicitly indicated in coding examples.

CHAPTER 1

INTRODUCTION TO VAX/VMS INPUT/OUTPUT

VAX/VMS supports a variety of input and output (I/O) devices, including disks, terminals, magnetic tapes, card readers, line printers, synchronous line interfaces, real-time I/O devices, and software mailboxes. This manual describes the capabilities of VAX/VMS device drivers and their programming interface, and gives several simple programming examples that use I/O drivers to perform input/output operations.

1.1 OVERVIEW OF VAX/VMS I/O

Input/output operations under VAX/VMS are designed to be as device- and function-independent as possible. User processes issue I/O requests to software channels, which form paths of communication with a particular device. Each process can establish its own correspondence between physical devices and channels. I/O requests are queued when they are issued and processed according to the relative priority of the process that issued them. I/O requests can be handled indirectly by the VAX-11 Record Management Services (RMS) or they can interface directly to the VAX/VMS I/O system. (VAX-11 RMS is described in the <u>VAX-11 Record Management Services Reference Manual</u>.)

To access the I/O services described in this manual, users issue system service requests. In certain system service requests, a function code included in the request defines the particular operation to be performed. For example, Queue I/O (QIO) system service requests can specify such operations as reading and writing blocks of data.

QIO requests can also specify a number of device-specific input/output operations, for example, converting lowercase characters to uppercase in terminal read operations or rewinding magnetic tape.

1.2 VAX/VMS I/O DEVICES

This manual describes VAX/VMS support for the following devices:

- Terminals, using the DZ11 Asynchronous Serial Line Multiplexer, and the VAX-11/780 console

- Disk devices:
  - RM03 Pack Disk
  - RP05 and RP06 Pack Disks
  - RK06 and RK07 Cartridge Disks
  - RX01 Floppy Disk
- Magnetic tape devices:
  - TE16 Magnetic Tape
  - TU45 and TU77 Magnetic Tape Systems
  - TS11 Magnetic Tape
- Line printers:
  - LP11 Line Printer Interface
  - LA11 DECprinter
- CR11 Card Reader
- DMC11 Synchronous Line Interface
- Mailboxes -- virtual devices used for interprocess transfer of information
- LPA11-K Laboratory Peripheral Accelerator
- DR32 Interface
- DUP11 Synchronous Line Interface

Chapters 2 through 8 and 10 through 12 describe in detail the drivers for these I/O devices and the I/O operations they perform.

1.3 SUMMARY OF I/O SYSTEM SERVICES

The following system services allow the direct use of the operating system's I/O resources:

- Assign I/O Channel ($ASSIGN)
- Deassign I/O Channel ($DASSGN)
- Queue I/O Request ($QIO)
- Queue I/O Request and Wait for Event Flag ($QIOW)
- Allocate Device ($ALLOC)
- Deallocate Device ($DALLOC)
- Get Channel Information ($GETCHN)
- Get Device Information ($GETDEV)
- Cancel I/O on Channel ($CANCEL)

- Create Mailbox and Assign Channel ($CREMBX)
- Delete Mailbox ($DELMBX)
- Wait for Single Event Flag ($WAITFR)
- Wait for Logical AND of Event Flags ($WFLAND)
- Wait for Logical OR of Event Flags ($WFLOR)
- Set AST Enable ($SETAST)
- Set Resource Wait Mode ($SETRWM)

This manual describes the use of system services for I/O operations. It also describes other system services used with I/O operations such as asynchronous system traps (ASTs) and event flag services. Section 1.8 describes the QIO request system service; ASTs and event flags, and $GETCHN are described in Sections 1.9 and 1.10, respectively. Section 1.8.7 describes the use of the $INPUT and $OUTPUT macros, which perform functions similar to the $QIOW system service.

See the VAX/VMS System Services Reference Manual for detailed information on all these system services and examples of their use. The VAX/VMS System Services Reference Manual also contains information on physical and logical device-naming conventions.

1.4 QUOTAS, PRIVILEGES, AND PROTECTION

To preserve the integrity of the system, VAX/VMS I/O operations are performed under the constraints of quotas, privileges, and protection.

Quotas establish a limit on the number and type of I/O operations that a process can perform concurrently. They ensure that all users have an equitable share of system resources and usage.

Privileges are granted to a user to allow the performance of certain I/O-related operations, for example, create a mailbox and perform logical I/O to a file-structured device. Restrictions on user privilege protect the integrity and performance of both the operating system and the services provided other users.

Protection is used to control access to files and devices. Device protection is provided in much the same way as file protection: shareable and nonshareable devices are protected by protection masks.

The Set Resource Wait Mode ($SETRWM) system service allows a process to select either of two modes when an attempt to exceed a quota occurs. In the enabled (default) mode, the process waits until the required resource is available before continuing. In the disabled mode, the process is notified immediately by a system service status return that an attempt to exceed a quota has occurred. Waiting for resources is transparent to the process when resource wait mode is enabled; no explicit action is taken by the process when a wait is necessary.

The different types of I/O-related quotas, privileges, and protection are described in the following paragraphs.

1.4.1 Buffered I/O Quota

The buffered I/O quota specifies the maximum number of concurrent buffered I/O operations a process can have active. In a buffered I/O operation, the user's data is buffered in system dynamic memory. The driver deals with the system buffer and not the user buffer. Buffered I/O is used for terminal, line printer, card reader, network, mailbox, and console medium (RX01) transfers. The user's buffer does not have to be locked in memory for a buffered I/O operation.

The buffered I/O quota value is established in the user authorization file by the system manager or by the process's creator. Resource wait mode is entered if enabled by the Set Resource Wait Mode system service and an attempt to exceed the buffered I/O quota is made.

1.4.2 Buffered I/O Byte Count Quota

The buffered I/O byte count quota specifies the maximum amount of buffer space that can be consumed from system dynamic memory for buffering I/O requests. All buffered I/O requests require system dynamic memory in which the actual I/O operation takes place.

The buffered I/O byte count quota is established in the user authorization file by the system manager or by the process's creator. Resource wait mode is entered if enabled by the Set Resource Wait Mode system service and an attempt to exceed the buffered I/O byte count quota is made.

1.4.3 Direct I/O Quota

The direct I/O quota specifies the maximum number of concurrent direct (that is, unbuffered), I/O operations that a process can have active. In a direct I/O operation, data is moved directly to or from the user buffer. Direct I/O is used for disk, magnetic tape, DMA real-time devices, and non-network DMC11 transfers. For direct I/O, the user's buffer must be locked in memory during the transfer.

The direct I/O quota value is established in the user authorization file by the system manager or by the process's creator. Resource wait mode is entered if enabled by the Set Resource Wait Mode system service and an attempt to exceed the direct I/O quota is made.

1.4.4 AST Quota

The AST quota specifies the maximum number of asynchronous system traps that a process can have outstanding. The quota value is established in the user authorization file by the system manager or by the process's creator. There is never an implied wait for this resource.

1.4.5 Physical I/O Privilege (PHY_IO)

Physical I/O privilege allows a process to perform physical I/O operations on a device. Physical I/O privilege also allows a process to perform logical I/O operations on a device. (Figures 1-1 and 1-2 show the use of physical I/O privilege in greater detail.)

1.4.6 Logical I/O Privilege (LOG_IO)

Logical I/O privilege allows a process to perform logical I/O operations on a device. A process can also perform physical operations on a device if the process has logical I/O privilege, the volume is mounted foreign, and the volume protection mask allows access to the device. (Figures 1-1 and 1-2 show the use of logical I/O privilege in greater detail.)

1.4.7 Mount Privilege

Mount privilege allows a process to use the IO$_MOUNT function to perform mount operations on disk and magnetic tape devices. IO$_MOUNT is used in ACP interface operations (see Chapter 9).

1.4.8 Volume Protection

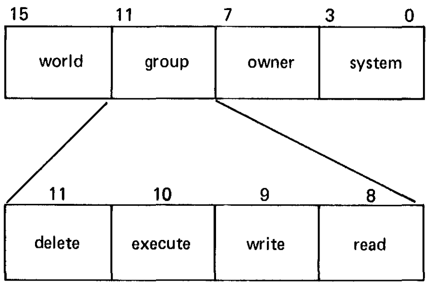
Volume protection protects the integrity of mailboxes and both foreign and Files-11 structured volumes. Volume protection for a foreign volume is established when the volume is mounted. Volume protection for a Files-11 structured volume is established when the volume is initialized. (The protection can be overridden when the volume is mounted if the process that is mounting the volume has the override volume protection privilege.)

Mailbox protection is established by the $CREMBX system service protection mask argument.

Protection for structured volumes and mailboxes is provided by a volume protection mask that contains four 4-bit fields. These fields correspond to the four classes of users that are permitted to access the volume. (User classes are based on the volume owner's user identification code, UIC.)

The 4-bit fields are interpreted differently for volumes that are mounted as structured (that is, volumes serviced by an Ancillary Control Process (ACP)) and volumes that are mounted as foreign.

The 4-bit fields have the following format for volumes mounted as structured:



The 4-bit fields have the following format for volumes mounted as foreign:

11	10	9	8
Log I/O	Phy I/O	*	*

*not used

Usually, volume protection is meaningful only for read and write operations.

1.4.9 Device Protection

Device protection protects the allocation of nonshareable devices, such as terminals and card readers.

Protection is provided by a device protection mask similar to that of volume protection, the difference being that only the bit corresponding to read access is checked and determines if the process can allocate or assign a channel to the device.

Device protection is established with the SET PROTECTION/DEVICE DCL operator command. Both the protection mask and the device owner UIC are set with this command.

1.4.10 System Privilege (SYSPRV)

System UIC privilege allows a process to be eligible for the volume or device protection specified for the system protection class, even though the process does not have a UIC in one of the system groups.

1.4.11 Bypass Privilege (BYPASS)

Bypass privilege allows a process to completely bypass volume and device protection.

1.5 SUMMARY OF VAX/VMS QIO OPERATIONS

VAX/VMS provides QIO operations that perform three basic I/O functions: read, write, and set mode. The read function transfers data from a device to a user-specified buffer. The write function transfers data in the opposite direction -- from a user-specified buffer to the device. For example, in a read QIO function to a terminal device, a user-specified buffer is filled with characters received from the terminal. In a write QIO function to the terminal, the data in a user-specified buffer is transferred to the terminal where it is displayed.

The set mode QIO function is used to control or describe the characteristics and operation of a device. For example, a set mode QIO function to a line printer can specify either uppercase or lowercase character format. Not all QIO functions are applicable to all types of devices. The line printer, for example, cannot perform a read QIO function.

1.6 PHYSICAL, LOGICAL, AND VIRTUAL I/O

I/O data transfers can occur in any one of three device addressing modes: physical, logical, or virtual. Any process with device access allowed by the volume protection mask can perform logical I/O on a device that is mounted foreign; physical I/O requires privilege. Virtual I/O does not require privilege; however, intervention by an ACP to control user access may be necessary if the device is under ACP control. (ACP functions are described in Chapter 9.)

1.6.1 Physical I/O Operations

In physical I/O operations, data is read from and written to the actual, physically addressable units accepted by the hardware; for example, sectors on a disk or binary characters on a terminal in the PASSALL mode. This mode allows direct access to all device-level I/O operations.

Physical I/O requires that one of the following conditions be met:

- The issuing process has physical I/O privilege (PHY_IO)

- The issuing process has logical I/O privilege (LOG_IO), the device is mounted foreign, and the volume protection mask allows physical access to the device

If neither of these conditions is met, the physical I/O operation is rejected by the QIO system service with a status return of SS$_NOPRIV (no privilege). Figure 1-1 illustrates the physical I/O access checks in greater detail.

The inhibit error-logging function modifier (IO$M_INHERLOG) can be specified for all physical I/O functions. IO$M_INHERLOG inhibits the logging of any error that occurs during the I/O operation.

1.6.2 Logical I/O Operations

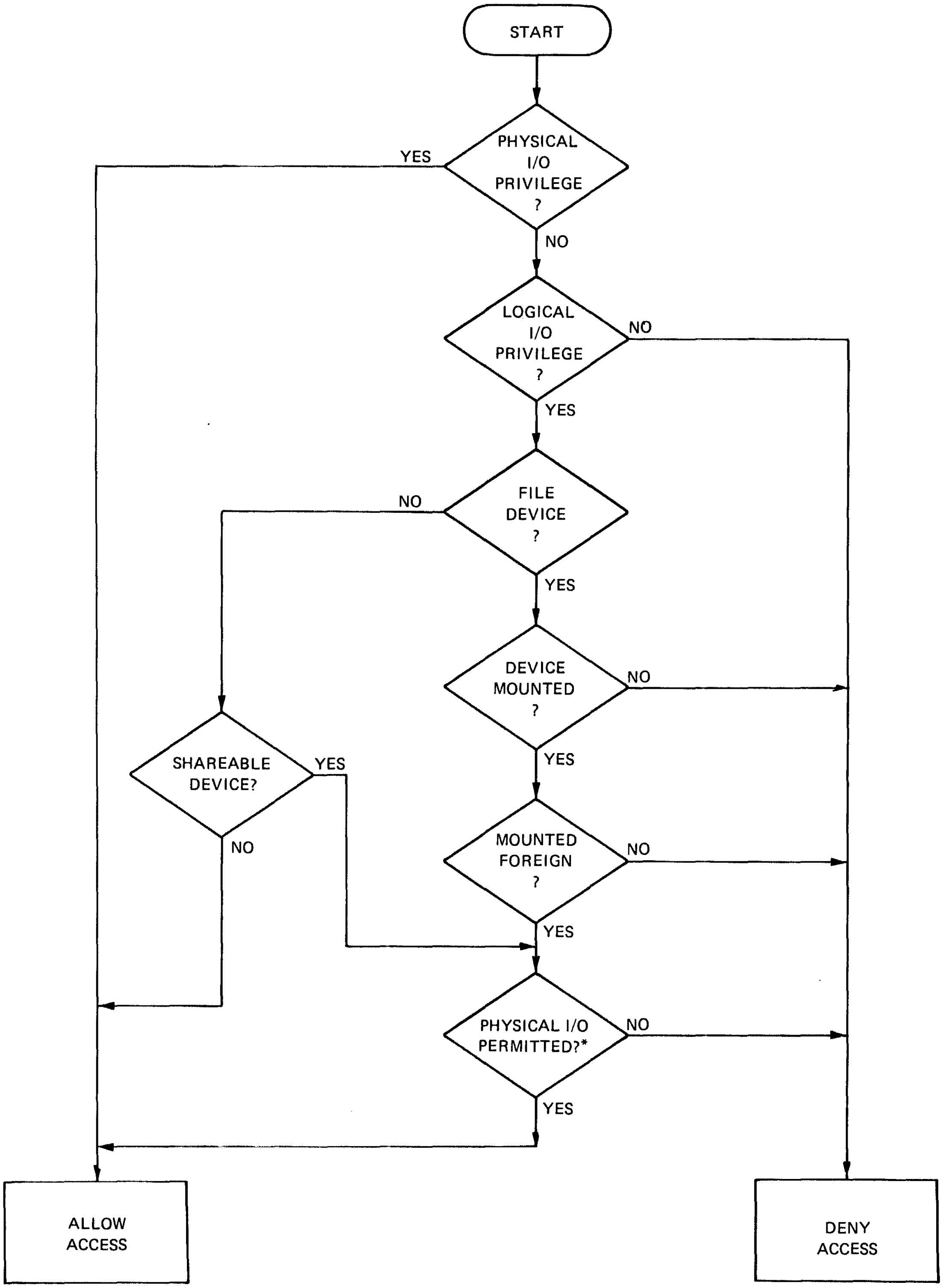
In logical I/O operations, data is read from and written to logically addressable units of the device. Logical operations can be performed on both block-addressable and record-oriented devices. For block-addressable devices (for example, disks), the addressable units are 512-byte blocks. They are numbered from 0 to n where n is the last block on the device. For record-oriented or non-block-structured devices (for example, terminals), logical addressable units are not pertinent and are ignored. Logical I/O requires that one of the following conditions be met:

- The issuing process has physical I/O privilege (PHY_IO)

- The issuing process has logical I/O privilege (LOG_IO)

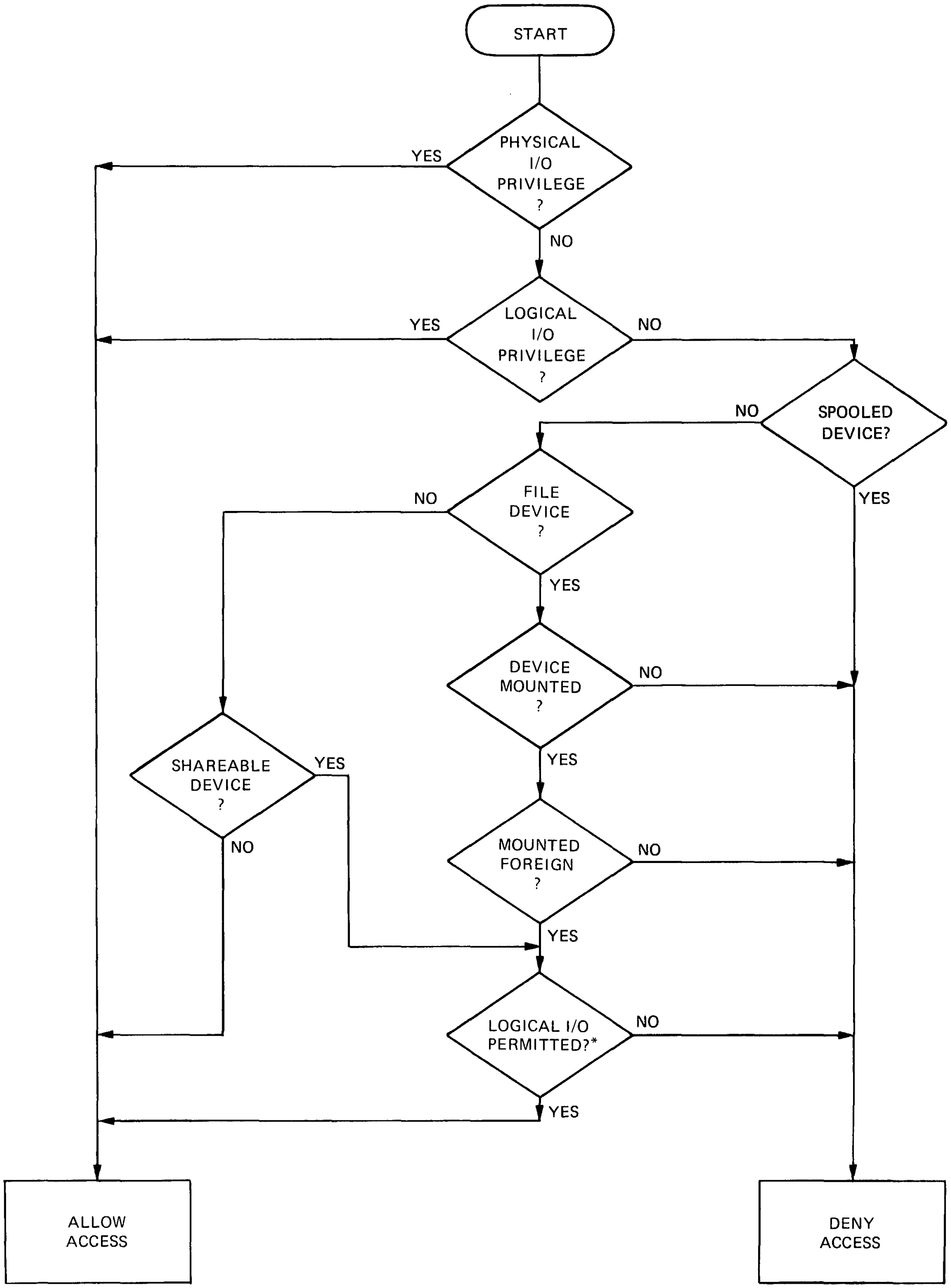
- The volume is mounted foreign and the volume protection mask allows access to the device

If none of these conditions is met, the logical I/O operation is rejected by the QIO system service with a status return of SS$_NOPRIV (no privilege). Figure 1-2 illustrates the logical I/O access checks in greater detail.



*Volume protection mask allows access

Figure 1-1 Physical I/O Access Checks



*Volume protection mask allows access

Figure 1-2 Logical I/O Access Checks

1.6.3 Virtual I/O Operations

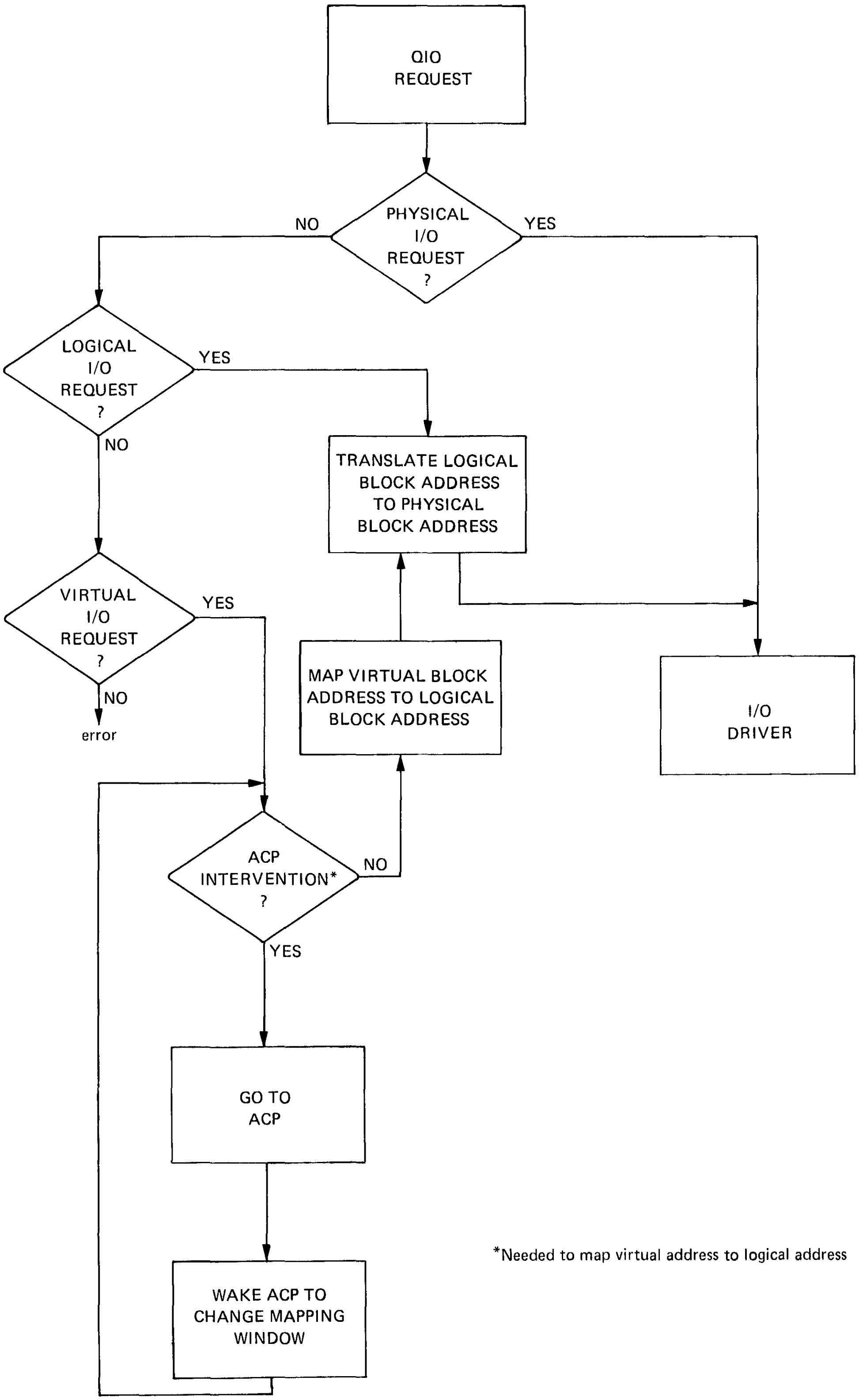
Virtual I/O operations can be performed on both record-oriented (non-file-structured) and block-addressable (file-structured) devices. For record-oriented devices (for example, terminals), the virtual function is the same as a logical function; the virtual addressable units of the devices are ignored.

For block-addressable devices (for example, disks), data is read from and written to open files. The addressable units in the file are 512-byte blocks. They are numbered starting at 1 and are relative to a file rather than to a device. Block-addressable devices must be mounted and structured and must contain a previously opened file.

Virtual I/O operations also require that the volume protection mask allow access to the device (a process having either physical or logical I/O privilege can override the volume protection mask). If these conditions are not met, the virtual I/O operation is rejected by the QIO system service with one of the following status returns:

Status Return	Meaning
SS$_NOPRIV	No privilege
SS$_DEVNOTMOUNT	Device not mounted
SS$_DEVFOREIGN	Volume mounted foreign (a foreign volume is a volume that does not contain a standard file structure understood by any of the VAX/VMS software)

Figure 1-3 shows the relationship of physical, logical, and virtual I/O to the driver.


QIO REQUEST
PHYSICAL I/O REQUEST ?
NO
YES
LOGICAL I/O REQUEST ?
YES
NO
TRANSLATE LOGICAL BLOCK ADDRESS TO PHYSICAL BLOCK ADDRESS
VIRTUAL I/O REQUEST ?
YES
NO
error
MAP VIRTUAL BLOCK ADDRESS TO LOGICAL BLOCK ADDRESS
I/O DRIVER
ACP INTERVENTION* ?
NO
YES
GO TO ACP
WAKE ACP TO CHANGE MAPPING WINDOW

*Needed to map virtual address to logical address

Figure 1-3 Physical, Logical, and Virtual I/O

1.7 I/O FUNCTION ENCODING

I/O functions fall into three groups that correspond to the three I/O device addressing modes (physical, logical, and virtual) described in Section 1.6. Depending on the device to which it is directed, an I/O function can be expressed in one, two, or all three modes.

I/O functions are described by 16-bit, symbolically-expressed values that specify the particular I/O operation to be performed and any optional function modifiers. Figure 1-4 shows the format of the 16-bit function value.

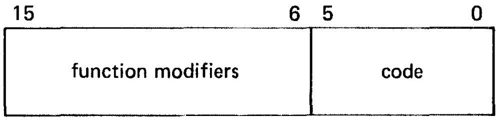


Figure 1-4 I/O Function Format

Symbolic names for I/O function codes are defined by the $IODEF macro, as described in the VAX/VMS System Services Reference Manual.

1.7.1 Function Codes

The low-order 6 bits of the function value are a code that specifies the particular operation to be performed. For example, the code for read logical block is expressed as IO$_READLBLK. Table 1-1 lists the symbolic values for read and write I/O functions in the three transfer modes.

Table 1-1
Read and Write I/O Functions

Physical I/O	Logical I/O	Virtual I/O
IO$_READPBLK IO$_WRITEPBLK	IO$_READLBLK IO$_WRITELBLK	IO$_READVBLK IO$_WRITEVBLK

The set mode I/O function has a symbolic value of IO$_SETMODE.

Function codes are defined for all supported devices. Although some of the function codes (for example, IO$_READVBLK and IO$_WRITEVBLK) are used with several types of devices, most are device dependent; that is, they perform functions specific to particular types of devices. For example, IO$_CREATE is a device-dependent function code; it is used only with file-structured devices such as disks and magnetic tapes. Chapters 2 through 8 and 10 through 12 provide complete descriptions of the functions and function codes.

1.7.2 Function Modifiers

The high-order 10 bits of the function value are function modifiers. These are individual bits that alter the basic operation to be performed. For example, the function modifier IO$M_NOECHO can be specified with the function IO$_READLBLK to a terminal. When used together, the two values are written as IO$_READLBLK!IO$M_NOECHO. This means that data typed at the terminal keyboard is entered in the user buffer but not echoed to the terminal. Figure 1-5 shows the format of function modifiers.

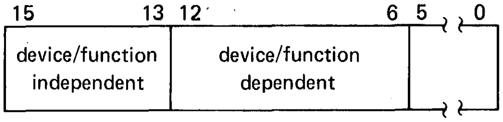


Figure 1-5 Function Modifier Format

As shown, bits 13 through 15 are device/function independent bits, and bits 6 through 12 are device/function dependent bits. Device/function dependent bits have the same meaning, whenever possible, for different device classes. For example, the function modifier IO$M_ACCESS is used with both disk and magnetic tape devices to cause a file to be accessed during a create operation. Device/function dependent bits always have the same function within the same device class.

There are two device/function independent modifier bits: IO$M_INHRETRY and IO$M_DATACHECK (a third bit is reserved). IO$M_INHRETRY is used to inhibit all error recovery. If any error occurs, and this modifier bit is specified, the operation is immediately terminated and a failure status is returned in the I/O status block (see Section 1.9.2). IO$M_DATACHECK is used to compare the data in memory with that on a disk or magnetic tape.

1.8 ISSUING I/O REQUESTS

This section describes the entire process involved in issuing I/O requests, including: assigning channels, allocating devices, and issuing QIO requests; the $QIO, $QIOW, $INPUT, and $OUTPUT macros; and, finally, status returns.

1.8.1 Channel Assignments

Before I/O requests can be made to a device, the user must assign a channel to establish a link between the user process and the device. A channel is a communication path associated with a device during VAX/VMS I/O operations. The process uses the channel to transfer information to and from the device.

The Assign I/O Channel ($ASSIGN) system service is used to assign a channel to a device. To code a call to the $ASSIGN system service, the user must supply the name of the device (physical device name or logical name) and the address of a word to receive the assigned channel number. The $ASSIGN system service returns the channel number. The process can then request an I/O operation by calling the Queue I/O ($QIO) system service and specifying, as one of the arguments, the channel number returned by the $ASSIGN system service.

In the following example, an I/O channel is assigned to the device TTB4. The channel number is returned in the word at TTCHAN.

```
     TTNAME:   .ASCID /_TTB4/          ;TERMINAL NAME DESCRIPTOR
     TTCHAN:   .BLKW  1                ;TERMINAL CHANNEL NUMBER
                  .
                  .
                  .
               $ASSIGN_S DEVNAM=TTNAME,CHAN=TTCHAN
```

If the first character in the device name (devnam) string is an underline character (_), the name is considered to be a physical device name; otherwise, one level of logical name translation is performed and the equivalence name, if any, is used.

The Create Mailbox and Assign Channel ($CREMBX) system service provides another way to assign a channel to a device. In this case, the device is a mailbox. $CREMBX creates a mailbox and then assigns a channel to it (see Section 7.1.1).

The QIO system service can be performed only on assigned I/O channels and only from access modes that are equal to or more privileged than the access mode from which the original channel assignment was made.

1.8.2 Device Allocation

A device can be allocated to a process (or subprocess) by the Allocate Device ($ALLOC) system service. The allocated device is reserved for the exclusive use of the requesting process, any subprocesses it creates, and subprocesses created by any related subprocess. No other process can allocate the device until the owning process explicitly deallocates it.

Channels can be assigned to both allocated and nonallocated devices; however, a process cannot assign a channel to a device that is allocated to another process. When a channel is assigned to a nonallocated, nonshareable device (for example, a line printer or a magnetic tape device) VAX/VMS implicitly allocates the device.

Access to device functions is controlled by physical and logical I/O privileges, the volume protection mask, the device protection mask, and the mountability of the device (a device is mountable if a MOUNT command can be issued for it). Even though a device is allocated to a process, the process cannot perform I/O operations on the device unless access is allowed.

1.8.3 I/O Function Requests

After a channel has been assigned, the process can request I/O functions by using the Queue I/O ($QIO) system service. The $QIO system service initiates an input or output operation by queuing a request to a specific device that is assigned to a channel.

Certain requirements must be met before a request is queued. For example, a valid channel number must be included in the request, the request must not exceed relevant quotas, and sufficient dynamic memory must be available to complete the operation. Failure to meet such requirements is indicated by a status return (described below in Section 1.8.8).

The number of pending I/O requests, the amount of buffer space, and the number of outstanding ASTs that a process can have are controlled by quotas.

Each I/O request causes an I/O request packet to be allocated from system dynamic memory. Additional memory is allocated under the following circumstances:

- The I/O request function is an ACP function
- The target device is a buffered I/O device
- The target device is a network I/O device

After an I/O request is queued, the system does not require the issuing process to wait for the I/O operation to complete. If the process that issued the QIO request cannot proceed until the I/O completes, an event flag can be used to synchronize I/O completion (see Sections 1.8.6.1 and 1.9.1). In this case, the process should request the Wait for Single Event Flag ($WAITFR) system service at the point where synchronization must occur: that is, where I/O completion is required.

$WAITFR specifies an event flag for which the process is to wait. (The $WAITFR event flag must have the same number as the event flag used in the QIO request.) The process then waits while the I/O operation is performed. On I/O completion, the event flag is set and the process is allowed to resume operation.

Other ways to achieve this synchronization include the use of the $QIOW system service and ASTs, described in Sections 1.8.5 and 1.9.3, respectively. In addition, the I/O status block can be specified and checked if the user wants to determine whether the I/O operation completed without an error, regardless of whether or not the process waits for I/O completion (see Section 1.9.2.)

The QIO system service is accompanied by up to six device/function-independent and six device/function-dependent arguments. Section 1.8.6 below describes device/function-independent arguments. The device/function-dependent arguments (P1 through P6) are potentially different for each device/function combination. However, similar functions that are performed by all devices have identical arguments. Furthermore, all functions performed by a particular class of device are identical. Device/function-dependent arguments are described in more detail for the individual devices in Chapters 2 through 8 and 10 through 12.

1.8.4 $QIO Macro Format

The general format for the $QIO macro, using position-dependent arguments, is:

```
$QIO_S   [efn],chan,func,[iosb],[astadr],[astprm],-
         [p1],[p2],[p3],[p4],[p5],[p6]
```

The first six arguments are device/function independent. If keyword arguments are used, they can be written in any order. Arguments P1 through P6 are device/function dependent. The chan and func arguments must be specified in each request; arguments enclosed in brackets ([]) are optional.

The following example illustrates a typical QIO request using keyword arguments:

```
$QIO_S      EFN=#1,-                    ;EVENT FLAG 1
            CHAN=TTCHAN1,-              ;CHANNEL
            FUNC=#IO$_WRITEVBLK,-       ;VIRTUAL WRITE
            P1=BUFADD,-                 ;BUFFER ADDRESS
            P2=#BUFSIZE                 ;BUFFER SIZE
```

1.8.5 $QIOW Macro Format

The Queue I/O Request and Wait For Event Flag ($QIOW) system service macro combines the $QIO and $WAITFR system services. It eliminates any need for explicit I/O synchronization by automatically waiting until the I/O operation is completed before returning control to the process. Thus, $QIOW provides a simpler way to synchronize the return to the originating process when the process cannot proceed until the I/O operation is completed.

The $QIOW macro has the same device/function independent and device/function dependent arguments as the $QIO macro:

```
$QIOW_S  [efn],chan,func,[iosb],[astadr],[astprm],-
         [p1],[p2],[p3],[p4],[p5],[p6]
```

1.8.6 $QIO and $QIOW Arguments

Table 1-2 lists the $QIO and $QIOW device/function-independent arguments and their meanings. Additional information is provided in the paragraphs following the table and in the VAX/VMS System Services Reference Manual.

Table 1-2
Device/Function-Independent Arguments

Argument	Meaning
efn (event flag number)	The number of the event flag that is to be cleared when the I/O function is queued and set when it is completed. This argument is optional in the macro form; if not specified, efn defaults to 0.
chan (channel number)	The number of the I/O channel to which the request is directed. The channel number is obtained from either the $ASSIGN or $CREMBX system service. This argument is mandatory in the macro form.
func (function value)	The 16-bit function code and modifier value that specifies the operation to be performed. This argument is mandatory in the macro form.

(continued on next page)

Table 1-2 (Cont.)
Device/Function-Independent Arguments

Argument	Meaning
iosb (I/O status block)	The address of a quadword I/O status block to receive the final I/O status. This argument is optional in the macro form.
astadr (AST address)	The entry point address of an AST routine to be asynchronously executed when the I/O completes. This argument is optional in the macro form.
astprm (AST parameter)	The 32-bit value to be passed to the AST routine as an argument when the I/O completes. It can be used to assist the routine in identifying the particular AST. This argument is optional in the macro form.

1.8.6.1 **Event Flag Number Argument** - The event flag number (efn) argument is the number of the event flag to be associated with the I/O operation. It is optional in a $QIO or $QIOW macro. The specified event flag is cleared when the request is issued and set when the I/O operation completes. The specified event flag is also set if the service terminates without queuing the I/O request.

If the process requested the $QIOW system service, execution is automatically suspended until the I/O completes. If the process requested the QIO system service (with no subsequent $WAITFR, $WFLOR, or $WFLAND macro), process execution proceeds in parallel with the I/O. As the process continues to execute, it can test the event flag at any point by using the Read Event Flags ($READEF) system service.

Event flag numbers must be in the range of 0 through 127 (however, event flags 24 through 31 are reserved for system use). If no specific event flag is desired, the efn argument can be omitted from the macro. In that case, efn defaults to 0.

Users should exercise care in the use of $QIOs and $QIOWs, for example, when a $QIOW is used for terminal input and a $QIO is used for terminal output. If no event flag is specified in either call, event flag 0 is set at the completion of the output $QIO and the waiting input $QIOW will prematurely return control to the process.

1.8.6.2 **Channel Number Argument** - The channel number (chan) argument represents the channel number of the physical device to be accessed by the I/O request. It is required for all $QIO and $QIOW requests. The association between the physical device and the channel is specific to the process issuing the I/O request. The channel number is obtained from the $ASSIGN or $CREMBX system service (as described above in Section 1.8.1).

1.8.6.3 **Function Argument** - The function (func) argument defines the logical, virtual, or physical I/O operation to be performed when the $QIO or $QIOW system service is requested. It is required for all QIO and QIOW requests. The argument consists of a 16-bit function code and function modifier. Up to 64 function codes can be defined. Function codes are defined for all supported device types; most of the codes are device dependent. The function arguments for each I/O driver are described in more detail in Chapters 2 through 8 and 10 through 12.

1.8.6.4 **I/O Status Block Argument** - The I/O status block (iosb) argument specifies the address of the I/O status block to be associated with the I/O request. It is optional in the QIO and QIOW macros. If omitted, the iosb value is 0 which indicates no iosb address is supplied. This block is a quadword that receives the final completion status of the I/O request. Section 1.9.2 describes the I/O status block in more detail.

1.8.6.5 **AST Address Argument** - The AST address (astadr) argument specifies the entry point address of an AST routine to be executed when the I/O operation is complete. If omitted, the astadr value is 0 which indicates no astadr address is supplied. This argument is optional and can be used to interrupt a process to execute special code at I/O completion. When the I/O operation completes, the AST service routine is CALLed at the address specified in the astadr argument. The AST service routine is then executed in the access mode from which the QIO service was called.

1.8.6.6 **AST Parameter Argument** - The AST parameter (astprm) argument is an optional, 32-bit arbitrary value that is passed to the AST service routine when I/O completes, to assist the routine in identifying the particular AST. A typical use of the astprm argument might be the address of a user control block. If omitted, the astprm value is 0.

1.8.6.7 **Device/Function-Dependent Arguments** - Up to six device/function-dependent arguments (P1 through P6) can be included in each QIO request. The arguments for terminal read function codes show a typical use of P1 through P6:

P1 = buffer address

P2 = buffer size

P3 = timeout count (for read with timeout)

P4 = read terminator descriptor block address

P5 = prompt string buffer address

P6 = prompt string buffer size

P1 is always treated as an address. Therefore, in the _S form of the macro, P1 always generates a PUSHAL instruction. P2 through P6 are always treated as values. In the _S form of the macro, these arguments always generate PUSHL instructions.

Inclusion of the device/function-dependent arguments in a QIO request depends on the physical device unit and the function specified. A user who wants to specify only a channel, an I/O function code, and an address for AST routine might issue the following:

```
     $QIO_S    CHAN=XYCHAN,FUNC=#IO$_READVBLK,-
               ASTADR=XYAST,P1=BUFADR,P2=#BUFLEN
```

In this example, XYCHAN is the address of the word containing the channel to which the request is directed; IO$_READVBLK is the function code; and XYAST is the AST entry point address. BUFADR and BUFLEN are the device/function-dependent arguments for an input buffer.

1.8.7 $INPUT and $OUTPUT Macro Format and Arguments

The $INPUT and $OUTPUT macros simplify the use of the $QIOW macro. These macros generate code to perform virtual operations, using the IO$_READVBLK and IO$_WRITEVBLK function codes (the function code is automatically specified in the request), and wait for I/O completion. The macro formats and arguments are:

```
     $INPUT    chan,length,buffer,[iosb],[efn]
     $OUTPUT   chan,length,buffer,[iosb],[efn]
```

Table 1-3 lists the $INPUT and $OUTPUT arguments and their meanings.

Table 1-3
$INPUT and $OUTPUT Arguments

Argument	Meaning
chan	The channel on which the I/O operation is to be performed.
length	The length of the input or output buffer.
buffer	The address of the input or output buffer.
iosb	The address of the quadword that receives the completion status of the I/O operation. This argument is optional.
efn	The number of the event flag for which the process waits. This argument is optional; if not specified, efn defaults to 0.

Both the iosb and efn arguments are optional; all other arguments must be included in each macro. Note that the order of the length and buffer arguments is opposite that of the $QIO and $QIOW P1 and P2 arguments. Also note that $INPUT and $OUTPUT do not have the astadr and astprm arguments; neither of these operations can conclude in an AST.

1.8.8 Status Returns for System Services

On completion of a system service call, the completion status is returned as a longword value in register R0, shown in Figure 1-6. (System services save the data in all registers except R0 and R1.)

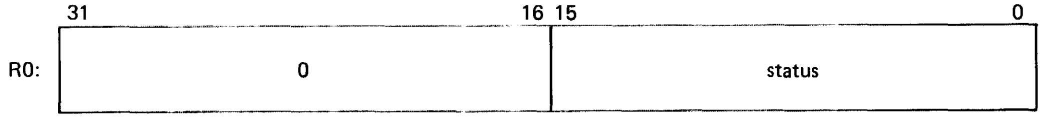


Figure 1-6 System Service Status Return

Completion status is indicated by a value in bits 0 through 15. The low-order 3 bits are encoded with the error severity level; all successful returns have an odd value:

0 =	warning
1 =	success
2 =	error
3 =	informational (nonstandard) success
4 =	severe error
5-7 =	reserved

Each numeric status code has a symbolic name in the form SS$_code. For example, the return might be SS$_NORMAL, which indicates successful completion of the system service. There are several error conditions that can be returned. For example, SS$_IVCHAN indicates that an invalid channel number was specified in an I/O request.

The VAX/VMS System Services Reference Manual describes the possible returns for each system service. Table 1-4 lists the valid status returns for the $QIO, $QIOW, $INPUT, and $OUTPUT system service requests.

Status returns for system services are not the same as the I/O status returns described in Chapters 2 through 8 and 10 through 12 for the various I/O drivers (see Section 1.9). A system service status return is the status of the $QIO, $QIOW, $INPUT, $OUTPUT, or other system service call after completion of the service, that is, after the system returns control to the user. A system service status return does not reflect the completion (successful or unsuccessful) of the requested I/O operation. For example, a $QIO system service read request to a terminal might be successful (status return is SS$_NORMAL) but fail because of a device parity error (I/O status return is SS$_PARITY). System service error status return codes refer only to failures to invoke the service.

An I/O status return is the status at the completion of the I/O operation. It is returned in the quadword I/O status block (IOSB). Although some of the symbolic names (for example, SS$_NORMAL and SS$_ACCVIO) can be used in both types of status returns, they have different meanings.

Table 1-4
$QIO, $QIOW, $INPUT, and $OUTPUT System Services Status Returns

Status	Meaning
SS$_NORMAL	The $QIO, $QIOW, $INPUT, or $OUTPUT request was successfully completed; that is, an I/O request was placed in the appropriate device queue.
SS$_ACCVIO	The IOSB, the specified buffer, or the argument list cannot be accessed by the caller.
SS$_EXQUOTA	The buffer quota, buffered I/O quota, or direct I/O quota was exceeded and the process has disabled resource wait mode with the $SETRWM system service. (The $SETRWM system service is described in Section 1.4.) SS$_EXQUOTA is also set if the AST quota was exceeded.
SS$_ILLEFC	An illegal event flag number was specified.
SS$_INSFMEM	Insufficient dynamic memory is available to complete the service and the process has disabled resource wait mode with the $SETRWM system service. (The $SETRWM system service is described in Section 1.4.)
SS$_IVCHAN	An invalid channel number was specified; that is, a channel number larger than the number of channels available.
SS$_NOPRIV	The specified channel was assigned from a more privileged access mode, the channel is not assigned, or the user does not have the proper privilege to access the device.
SS$_UNASEFC	A common event flag in an unassociated event flag cluster was specified.
SS$_ABORT	A network logical link was broken.
SS$_INSFARG	Not enough arguments were supplied to the service.
SS$_ILLSER	An invalid system service was called.

1.9 I/O COMPLETION

Whether an I/O request completed successfully or unsuccessfully can be denoted by one or more return conditions. The selection of the return conditions depends on the arguments included in the QIO macro call. The three primary returns are:

- Event flag - an event flag is set on completion of an I/O operation.

- I/O status block - if the iosb argument was specified in the QIO macro call, a code identifying the type of success or failure is returned in bits 0 through 15 of a quadword I/O status block on completion of the I/O operation. The location of this block is indicated by the user-supplied iosb argument.

- Asynchronous system trap - if an AST address argument was specified in the I/O request, a call to the AST service routine occurs, at the address indicated, on completion of the I/O operation. (The I/O status block, if specified in the I/O request, is updated prior to the AST call.)

1.9.1 **Event Flags**

Event flags are status posting bits used by the $QIO, $QIOW, $INPUT, and $OUTPUT system services to indicate the completion or occurrence of an event. The system service clears the event flag when the operation is queued and sets it when the operation is completed. Event flag services allow users to set or clear certain flags, test the current status of flags, or place a program in a wait state pending the setting of a flag or group of flags.

See the VAX/VMS System Services Reference Manual for more information on event flags and their use.

1.9.2 **I/O Status Block**

The completion status of an I/O request is returned in the first word of the I/O status block (IOSB), as shown in Figure 1-7.

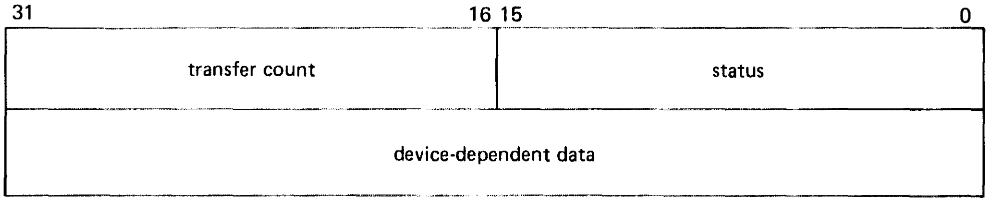


Figure 1-7 I/O Status Block Format

The IOSB indicates whether the operation was successfully completed, the amount of data transferred, and additional device-dependent information such as the number of lines printed. The status return code has the same format and bit significance (bit 0 set indicates success; bit 0 clear indicates error) as the system service status code (see Section 1.8.8). For example, if the process attempts to access a nonexistent disk, a status code of SS$_NONEXDRV is returned in the I/O status block. The status returns for the individual I/O drivers are listed in Chapters 2 through 8 and 10 through 12.

The upper half of the first IOSB longword contains the transfer count on completion of the I/O operation if the operation involved the transfer of data to or from a user buffer. For example, if a read operation is performed on a terminal, the number of bytes typed before a carriage return is indicated here. If a magnetic tape unit is the device and a read function is specified, the transfer count represents the number of bytes actually transferred. The second longword of the IOSB can contain certain device-dependent information. This information is supplied in more detail for each I/O driver in Chapters 2 through 8 and 10 through 12.

The status can be tested symbolically, by name. For example, the SS$_NORMAL status is returned if the operation was completed successfully. The following example illustrates the examination of the I/O status block XYIOSB to determine if an error occurred:

```
$QIO_S    CHAN=XYCHAN,FUNC=#IO$_WRITEVBLK,-
          IOSB=XYIOSB,P1=BUFADR,P2=#BUFLEN
BLBC      R0,REQERR                       ;CHECK SYSTEM SERVICE
                                          ;STATUS CODE
          .
          .
          .
CMPW      #SS$_NORMAL,XYIOSB              ;CHECK I/O STATUS
                                          ;CODE
BNEQ      ERROR
```

The status block can be omitted from a QIO request if the user wishes to assume successful completion of the request and does not want to know how many bytes were transferred. If specified, the IOSB is cleared when the QIO request is issued and then filled with the final status at I/O completion.

1.9.3 Asynchronous System Traps

As an option, an AST routine can be specified in the QIO request if the user wants to interrupt the normal execution of a process to execute special code on completion of the request. Even if the process is blocked for a $WAITFR or $QIOW, it will be interrupted. When the I/O operation completes, a CALL instruction is used to transfer control to the AST service routine at the entry point address specified in the QIO astadr argument. The address must be the address of an entry mask. The AST service routine is then executed at the access mode from which the QIO request was issued. Figure 1-8 shows the argument list for the CALL instruction.

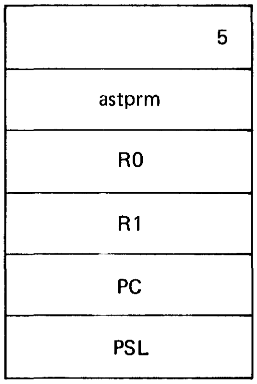


Figure 1-8 CALL Instruction Argument List

Using an AST to signal I/O completion allows the process to be occupied with other functions during the I/O operation. The process need not wait until some event occurs before proceeding to another operation.

See the VAX/VMS System Services Reference Manual for more detailed information on ASTs and their use.

1.10 DEVICE INFORMATION

Two system services can be used to obtain information about devices: Get Channel Information ($GETCHN) and Get Device Information ($GETDEV) system services. The information obtained includes such categories as device characteristics, device type, error count, and operation count.

The Get Channel Information ($GETCHN) system service is used to obtain information about a device to which an I/O channel is assigned. The $GETCHN system service can be performed only on assigned channels and only from access modes that are equal to, or more privileged than, the access mode from which the original channel assignment was made.

The Get Device Information ($GETDEV) system service is used to obtain information about any device.

$GETCHN and $GETDEV return both primary and secondary device characteristics. Usually, these characteristics are identical. However, they can differ in three instances:

1. If the device is a spooled device, the primary characteristics are those of the intermediate device and the secondary characteristics are those of the spooled device. See the VAX/VMS System Manager's Guide for information on spooling.

2. If the device represents a logical link on a network, the secondary characteristics contain information about the link.

3. If the device has an associated mailbox, the primary characteristics are those of the device and the secondary characteristics are those of the mailbox.

The macro format for a $GETCHN request is:

```
     $GETCHN chan,[prilen],[pribuf],[scdlen],[scdbuf]
```

The macro format for a $GETDEV request is:

```
     $GETDEV devnam,[prilen],[pribuf],[scdlen],[scdbuf]
```

Table 1-5 lists the $GETCHN and $GETDEV arguments and their meanings.

Table 1-5
$GETCHN and $GETDEV Arguments

Argument	Meaning
chan	The number of the I/O channel to which a $GETCHN request is directed (this is not an argument for $GETDEV).
devnam	The address of a string descriptor for the name of the device to which $GETDEV is directed (this is not an argument for $GETCHN).
prilen	The address of the word to receive the length of the primary characteristics. This argument is optional.

(continued on next page)

Table 1-5 (Cont.)
$GETCHN and $GETDEV Arguments

Argument	Meaning
pribuf	The address of the buffer descriptor for the buffer that is to receive the primary device characteristics. An address of 0 indicates that no buffer is specified. This argument is optional.
scdlen	The address of the word to receive the length of the secondary characteristics. This argument is optional.
scdbuf	The address of the buffer descriptor for the buffer that is to receive the secondary device characteristics. An address of 0 indicates that no buffer is specified. This argument is optional.

Figure 1-9 shows the format of the device information returned in the primary and secondary buffers.

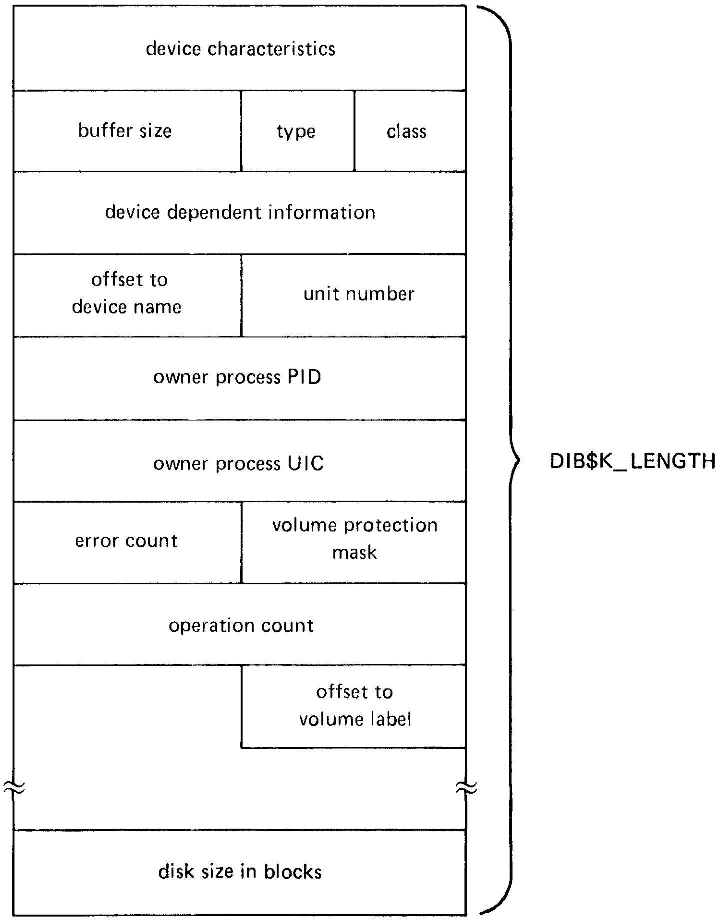


Figure 1-9 Buffer Format for $GETCHN and $GETDEV System Services

In Figure 1-9, offsets are the displacement from the beginning of the buffer to the specified field. Missing fields are denoted by offsets of 0. Both device name and volume label are stored in the buffer as counted strings. They must be located through the use of their respective offset values. Symbolic offsets for all fields are defined by the $DIBDEF macro. If both a volume label and a device name are returned, the buffer has a length of 64 bytes.

As much information as possible is returned for each of the primary and secondary characteristics. If all the information does not fit in the specified buffers, an appropriate status value is returned. Table 1-6 lists the status return values for the $GETCHN and $GETDEV system services.

Table 1-6
$GETCHN and $GETDEV Status Returns

Status	Meaning
SS$_NORMAL	The $GETCHN or $GETDEV system service successfully completed.
SS$_ACCVIO	The caller cannot read a buffer descriptor, write a buffer, or access the argument list.
SS$_IVCHAN	An invalid channel number was specified in the $GETCHN request, that is, a channel number larger than the number of channels available; the channel is nonexistent.
SS$_NOPRIV	The caller does not have the privilege to access the specified channel or the channel is unassigned.
SS$_BUFFEROVF	The $GETCHN or $GETDEV system service successfully completed. The device information returned overflowed the buffer(s) provided and has been truncated.

CHAPTER 2

TERMINAL DRIVER

This chapter describes the use of the VAX/VMS terminal driver. This driver supports the DZ-11 Asynchronous Serial Line Multiplexer and the console terminal.

2.1 SUPPORTED TERMINAL DEVICES

Each DZ-11 multiplexer interfaces 8 or 16 asynchronous serial communication lines for use with terminals. It supports programmable baud rates; however, input and output speeds must be the same. VAX/VMS supports the DZ-11 internal modem control.

The system console terminal is attached to the processor with a special purpose interface.

The Remote (DECnet) Command Terminal also makes use of the features and capabilities listed in Section 2.2.

2.2 TERMINAL DRIVER FEATURES AND CAPABILITIES

The VAX/VMS terminal driver provides the following capabilities:

- Type-ahead
- Specifiable or default line terminators
- Special operating modes, such as NOECHO and PASSALL
- American National Standard escape sequence detection
- Terminal/mailbox interaction
- Terminal control characters and special keys
- Dial-up
- Optional parity specification
- Limited full-duplex operation (simultaneously active read and write requests)

2.2.1 Type-ahead

Input (data received) from a VAX/VMS terminal is always independent of concurrent output (data sent) to a terminal. This capability is called type-ahead. Type-ahead is allowed on all terminals unless explicitly disabled by the Set Mode characteristic, inhibit type-ahead (TT$M_NOTYPEAHD; see Section 2.4.3).

Data typed at the terminal is retained in the type-ahead buffer until the user program issues an I/O request for a read operation. At that time, the data is transferred to the program buffer and echoed at the terminal where it was typed.

Deferring the echo until the read operation is active allows the user process to specify function code modifiers that modify the read operation. These modifiers can include, for example, noecho (IO$M_NOECHO) and convert lowercase characters to uppercase (IO$M_CVTLOW) (see Section 2.4.1.1).

If a read operation is already in progress when the data is typed at the terminal, the data transfer and echo are immediate.

The action of the driver when the type-ahead buffer fills depends on the Set Mode characteristic TT$M_HOSTSYNC (see Section 2.4.3). If TT$M_HOSTSYNC is not set, CTRL/G (BELL) is returned to inform the user that the type-ahead buffer is full. If TT$M_HOSTSYNC is set, the driver stops input by sending a CTRL/S and the terminal responds by sending no more characters. These warning operations are begun 8 characters before the type-ahead buffer fills. The driver sends a CTRL/Q to restart transmission when the type-ahead buffer empties completely.

The type-ahead buffer length is variable, with possible values in the range from 0 through 32,767. The length can be set on a system-wide basis through use of the system generation parameter TTY_TYPAHDSZ.

2.2.2 Line Terminators

A line terminator is the control sequence that the user types at the terminal to indicate the end of an input line. Optionally, the user process can specify a particular line terminator or class of terminators for read operations.

Terminators are specified by an argument to the QIO request for a read operation. By default, they can be any ASCII control character except FF, VT, LF, TAB, or BS. If included in the request, the argument is a user-selected group of characters (see Section 2.4.1.2).

All characters are 7-bit ASCII characters unless data is input on an 8-bit terminal (see Section 2.4.1). (The characteristic TT$M_EIGHTBIT determines whether a terminal uses the 7-bit or 8-bit character set; see Table 2-4.) All input characters are tested against the selected terminator(s). The input is terminated when a match occurs or the user's input buffer fills.

2.2.3 Special Operating Modes

The VAX/VMS terminal driver supports many special operating modes for terminal lines. Section 2.4.3 lists these modes. All special modes are enabled or disabled by the Set Mode and Set Characteristics QIOs.

2.2.4 Escape Sequences

Escape sequences are strings of two or more characters, beginning with the escape character (decimal 27 or hexadecimal 1B), that indicate that control information follows. Many terminals send and respond to such escape sequences to request special character sets or to indicate the position of a cursor.

The Set Mode characteristic TT$M_ESCAPE (see Section 2.4.3) is used to specify that VAX/VMS terminal lines can generate valid escape sequences. If this characteristic is set, the terminal driver verifies the syntax of the escape sequences. The sequence is always considered a read function terminator and is returned in the read buffer, that is, a read buffer can contain other characters that are not part of an escape sequence, but a complete escape sequence always terminates a read. The return information in the read buffer and the I/O status block includes the position and size of the terminating escape sequence in the data record (see Section 2.5).

Any escape sequence received from a terminal is checked for correct syntax. If the syntax is not correct, SS$_BADESCAPE is returned as the status of the I/O. If the escape sequence does not fit in the user buffer, SS$_PARTESCAPE is returned. The remaining characters are transmitted on the next read. No syntax integrity is guaranteed across read operations. Escape sequences are never echoed. Valid escape sequences are any of the following forms (hexadecimal notation):

```
ESC <int>...<int> <fin>
```

where:

ESC is pressing the ESC key, a byte (character) of 1B

<int> is an "intermediate character" in the range of 20 to 2F. This range includes the character "space" and 15 punctuation marks. An escape sequence can contain any number of intermediate characters, or none.

<fin> is a "final character" in the range of 30 to 7E. This range includes uppercase and lowercase letters, numbers, and 13 punctuation marks.

There are four additional escape sequence forms:

```
ESC <;> <20-2F>...<30-7E>
ESC <?> <20-2F>...<30-7E>
ESC <O> <20-2F>...<40-7E>
ESC <Y> <20-7E>...<20-7E>
```

For example, when the IDENTIFY escape sequence, escape Z, is sent to a VT-55 terminal, the response from the terminal is ESC <C>. (Escape sequences are neither displayed nor echoed on the terminal.)

Control sequences, as defined by the ANSI standard, are escape sequences which include control parameters. Control sequences have the following format:

```
ESC [ <par>...<par> <int>...<int> <fin>
```

where:

ESC   is pressing the ESC key

[denotes a control sequence

<par>   is a parameter specifier in the range of 30 to 3F

<int>   is an "intermediate character", as defined for escape sequences

<fin>   is a "final character" in the range of 40 to 7E

For example, the position cursor control sequence is:

```
ESC [ Pl ;  Pc H
```

where Pl is the desired line position and Pc is the desired column position.

The VT100 User Guide lists valid escape sequences for VT100 terminals.

Section 2.2.6 describes control character functions during escape sequences.

2.2.5 Terminal/Mailbox Interaction

Mailboxes are virtual I/O devices used for communication between processes. The terminal I/O driver can use a mailbox to communicate with a user process. Chapter 7 describes the mailbox driver.

A user program can use the $ASSIGN system service to associate a mailbox with one or more terminals. The terminal driver sends messages to this mailbox when terminal-related events occur that require the attention of the user image.

Mailboxes used in this way carry status messages, not terminal data, from the driver to the user program. For example, when data is received from a terminal for which no read request is outstanding (unsolicited data), a message is sent to the associated mailbox to indicate data availability. On receiving this message, the user program must read the channel assigned to the terminal to obtain the data. Messages are sent to mailboxes under the following conditions:

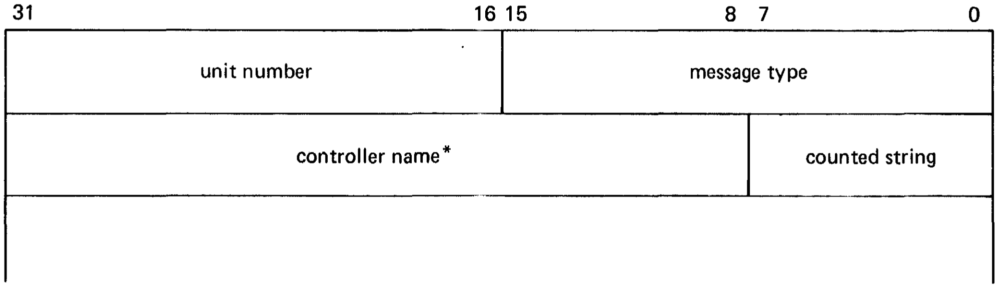
- Unsolicited data in the type-ahead buffer. The use of the associated mailbox can be enabled and disabled as a subfunction of the read and write requests (see Sections 2.4.1 and 2.4.2). Thus, the user process can enter into a dialogue with the terminal after an unsolicited data message arrives. Then, after the dialogue is over, the user process can reenable the unsolicited data message function on the last I/O exchange. The default on all terminals is enabled. Only one message is sent between read operations.

- Terminal hang-up. Hang-up occurs when a remote line loses the carrier signal; a message is sent to the mailbox. When hang-up occurs on lines that have the characteristic TT$M_REMOTE set, the line returns to local mode.

Messages placed in the mailbox have the following content and format:

- Message type. The codes MSG$_TRMUNSOLIC (unsolicited data) and MSG$_TRMHANGUP (hang-up) identify the type of message. Message types are identified by the $MSGDEF macro.
- Device unit number to identify the terminal that sent the message.
- Counted string to specify the device name.
- Controller name

Figure 2-1 illustrates this format.



*does not include the colon (:) character

Figure 2-1 Terminal Mailbox Message Format

Interaction with a mailbox associated with a terminal occurs through standard QIO functions and ASTs. Therefore, the process need not have outstanding read requests to an interactive terminal to respond to the arrival of unsolicited data. The process need only respond when the mailbox signals the availability of unsolicited data. Section 2.6 contains an example of mailbox programming.

The ratio of terminals to mailboxes is not always one to one. One user process can have many terminals associated with a single mailbox.

2.2.6 Control Characters and Special Keys

A control character is a character that controls action at the terminal rather than passing data to a process. An ASCII control character has a code between 0 and 31, plus 126 and 127 (hexadecimal 0 through 1F, plus 7E and 7F), that is, all normal characters plus DELETE and ALTMODE. Some control characters are typed at the terminal by simultaneously pressing the CTRL key and a character key, that is, CTRL/x. Other control characters, for example, RETURN, LINE FEED, and ESCAPE, are typed by pressing a single key, that is, RET, LF, and ESC. Table 2-1 lists the VAX/VMS terminal control characters (none of these characters is interpreted in the PASSALL mode). Table 2-2 lists special terminal keys.

Table 2-1
Terminal Control Characters

Control Character	Meaning
CTRL/C	Gains the attention of the enabling process if the user program has enabled a CTRL/C AST. If a CTRL/C AST is not enabled, CTRL/C is converted to CTRL/Y (see Section 2.4.3). If echo is not disabled, the terminal performs a newline (carriage return followed by a line feed), types ^C, and performs another newline. Additional consequences of CTRL/C are: • The type-ahead buffer is flushed. • CTRL/S and CTRL/O are reset. • The current I/O operation (if any) is successfully completed. The status return is SS$_CONTROLC, or SS$_CONTROLY if CTRL/C is converted to CTRL/Y.
TAB (CTRL/I)	Tabs horizontally. Advances to the next tab stop on terminals with the characteristic TT$M_MECHTAB, but the driver assumes tab stops on MODULO 8, that is, multiples of 8, cursor positions. On terminals without this characteristic, enough spaces are output to move the cursor to the next MODULO (8) position.
LF (CTRL/J)	Performs line feed; filled if TT$M_LFFILL is set.
VT (CTRL/K)	Terminal performs a vertical tab.
FF (CTRL/L)	Performs form feed. Advances to the top of the page on terminals with the characteristic TT$M_MECHFORM. On terminals without this characteristic, the driver sends enough LF (filled) to move the paper to the top of form position described by the length of the page and the current position of the page. The driver then sends a carriage return to position the cursor at the left margin. The Set Mode or Set Characteristics functions can be used to set page length (see Section 2.4.3).
CTRL/O	Discards output. Action is immediate. All output is discarded until the next read operation, the next write operation with a IO$M_CANCTRLO modifier, or the receipt of the next CTRL/O. If echo is not disabled, the terminal echoes ^O, followed by a newline. The current write operation (if any) and write operations performed while CTRL/O is in effect are completed with a status return of SS$_CONTROLO. CTRL/O, which reenables output, cancels CTRL/S. CTRL/C and CTRL/Y cancel CTRL/O.

(continued on next page)

Table 2-1 (Cont.)
Terminal Control Characters

Control Character	Meaning
CTRL/Q	Controls data flow; used by terminals and the driver. Restarts data flow to and from a terminal if previously stopped by CTRL/S. The action occurs immediately with no echo. CTRL/Q is also used to solicit read operations. CTRL/Q is meaningless if the line does not have the characteristic TT$M_TTSYNC, the characteristic TT$M_HOSTSYNC, the characteristic TT$M_READSYNC, or is not currently stopped by CTRL/S.
CTRL/R	Displays current input. When CTRL/R is typed during a read operation, a newline is echoed, and the current contents of the input buffer is displayed. If the current operation is a read with prompt (IO$_READPROMPT) operation, the current prompt string is also displayed. CTRL/R has no effect if the characteristic TT$M_NOECHO is set.
CTRL/S	Controls data flow; used by both terminals and the driver. CTRL/S stops all data flow; the action occurs immediately with no echo. CTRL/S is also used to stop read operations. CTRL/S is meaningful only if the terminal has the TTM_TTSYNC, TTM_HOSTSYNC, or TT$M_READSYNC characteristic.
CTRL/U	Purges current input data. When CTRL/U is typed before the end of a read operation, the current input is flushed. If echo is not disabled, the terminal echoes ^U, followed by a newline. The prompt string is displayed again if the current operation is a read with prompt (IO$_READPROMPT). CTRL/U has no effect on type-ahead buffer operations.
CTRL/X	Purges the type-ahead buffer and performs a CTRL/U operation. Action is immediate. If a read operation is in progress, the terminal echoes ^U, followed by a newline.
CTRL/Y	CTRL/Y is a special interrupt or attention character that is used to gain the attention of the command interpreter for a logged-in process. CTRL/Y can be enabled with an IO$M_CTRLYAST function modifier to a IO$_SETCHAR or IO$_SETMODE QIO. Physical or logical I/O privilege, or an access mode greater than that held by the user, is required to enable CTRL/Y ASTs. The command interpreter's CTRL/Y AST handler always has precedence over a user program's CTRL/Y AST handler.

(continued on next page)

Table 2-1 (Cont.)
Terminal Control Characters

Control Character	Meaning
CTRL/Y (Cont.)	Typing CTRL/Y results in an AST to an enabled process process to signify that the user typed CTRL/Y. The terminal performs a newline, types ^Y, and performs another newline if the AST and echo are enabled. CTRL/Y is ignored (and not echoed) if the process is not enabled for the AST. Additional consequences of CTRL/Y are: • The type-ahead buffer is flushed. • CTRL/S mode is reset. • The current I/O operation (if any) is successfully completed with a 0 transfer count. The status return is SS$_CONTROLY.
CTRL/Z	Echoes ^Z when CTRL/Z is typed as a read terminator. By convention, CTRL/Z constitutes end-of-file.

Table 2-2
Special Terminal Keys

Control Character	Meaning
ALTMODE	(Decimal 126 or hexadecimal 7E) Converts to escape on terminals that do not have the lowercase characteristic TT$M_LOWER set.
DEL (DELETE)	(Decimal 127 or hexadecimal 7F) Removes last typed character from input stream. DEL is ignored if there are currently no input characters. Hard copy terminals echo the deleted character enclosed in backslashes. For example, if the character z is deleted, \z\ is echoed (the second backslash is echoed after the next non-DEL character is typed). CRT terminals echo DEL as a backspace followed by a space and another backspace.
ESC (ESCAPE)	If escape sequences are recognized (the Set Mode characteristic TT$M_ESCAPE is set), pressing ESC signals the beginning of an escape sequence. On these terminals ESC is never echoed; however, on terminals that do not recognize escape sequences, ESC is echoed as a dollar sign ($) if it was used as a read terminator or as hexadecimal 1B if it was not a read terminator.
RET (RETURN)	If used during a read (input) operation, RET echoes a newline. All returns are filled on terminals with TT$M_CRFILL specified.

2.2.7 Dial-Up

VAX/VMS supports the DZ-11 internal modem control (for example, Bell 103A, Bell 113, or equivalent) in autoanswer, full duplex mode. The terminal driver does not support half-duplex operations on modems such as the Bell 202. Also not supported are modems that use circuit 108/1 (connect data set to line) in place of data terminal ready. All United States modems and most European modems use the data terminal signal which is supported. The terminal characteristic TT$M_REMOTE designates the line as being remote to the local computer. The driver automatically sets TT$M_REMOTE if the carrier signal changes from off to on.

Dial-up lines are monitored periodically to detect a change in the modem carrier signal. The system generation parameter TTYSCANDELTA establishes the dial-up monitoring period (see the VAX-11 Software Installation Guide).

If a line's carrier signal is lost, the driver waits nine monitor periods for the carrier signal to return or none if the system generation parameter DIALTYPE is 1. (DIALTYPE is 0 by default and is relevant to the United Kingdom only.) If the carrier signal is not detected during this time, the line is "hungup." The hang-up action signals the owner of the line, through a mailbox message, that the line is no longer in use. (No dial-in message is sent; the unsolicited character message is sufficient when the first available data is received.) The line is not available for two monitor periods after the hang-up sequence begins. The hang-up sequence is not reversible. If the line hangs up, all enabled CTRL/Y ASTs are delivered; the CTRL/Y AST P2 argument is overwritten with SS$_HANGUP. The I/O operation in progress is cancelled and the status value SS$_ABORT is returned in the IOSB.

When a line with the TT$M_REMOTE characteristic is hung up, the characteristics of the line return to TT$M_LOCAL.

2.2.8 Duplex Modes

The VAX/VMS terminal driver can execute in either half- or full-duplex mode. These terms describe the terminal driver software, not the terminal communication lines. For the communication lines, the driver supports the DZ-11 Asynchronous Serial Line Multiplexer and the console terminal. In terminal driver software, the terms half- and full-duplex refer to ordering algorithms used to service read and write requests.

In half-duplex mode, all read and write requests are inserted onto one queue. The driver removes requests from the head of this queue and executes them one at a time; all requests are sequentially executed in the order they were issued.

In full-duplex mode, read requests are inserted onto one queue and write requests onto another. The existence of two queues allows the driver to recognize the presence of two requests -- a read and a write -- at the same time. However, the driver cannot execute a read request and a write request simultaneously. When it is ready to service a request, the driver dynamically decides which request -- the read or the write -- to process next.

In the VAX/VMS terminal driver, write requests normally have priority. A write request can interrupt a current, but inactive, read request. A read request is current when the terminal driver removes that request from the head of the read queue. In a simple read operation, the read request becomes active when the first input character is received and echoed. In a read with prompt operation, the read request becomes active when the first character of the prompt is output to the terminal. Once a read request becomes active, all write requests will be queued until the read completes. However, during a simple read operation many write requests can be executed before the first input character is typed at the terminal.

When all I/O requests are issued using $QIOW calls, there can be only one current I/O request at any time. In this case, the order in which requests are serviced is the same for both half- and full-duplex modes.

The type ahead buffer always buffers input data for which there is no current read request, in both half- and full-duplex modes.

2.3 DEVICE INFORMATION

The user process can obtain terminal characteristics by using the $GETCHN and $GETDEV system services (see Section 1.10). The terminal-specific information is returned in the first three longwords of a user-specified buffer as shown in Figure 2-2 (Figure 1-9 shows the entire buffer).

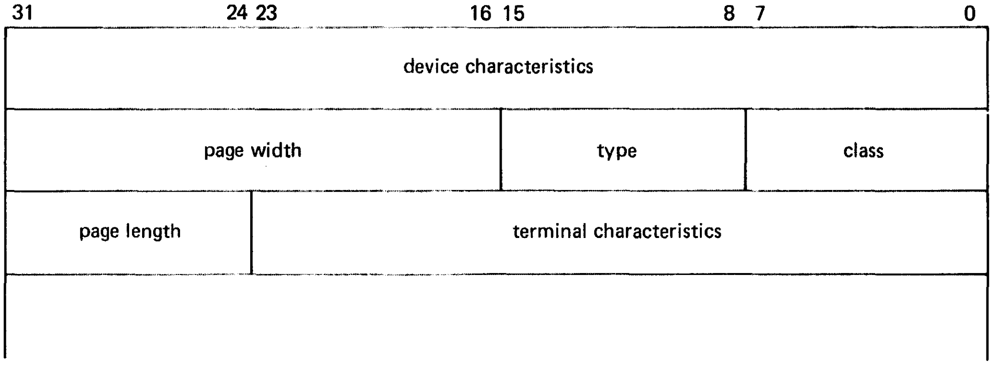


Figure 2-2 Terminal Information

The first longword contains device-independent data. The second and third longwords contain device-dependent data.

Table 2-3 lists the device-independent characteristics returned in the first longword.

Table 2-3
Terminal Device-Independent Characteristics

Characteristic Name[1]	Meaning
DEV$M_AVL	Terminal is on line and available
DEV$M_IDV	Terminal is capable of input
DEV$M_ODV	Terminal is capable of output
DEV$M_SPL	Spooled
DEV$M_CCL	Carriage control
DEV$M_REC	Record oriented
DEV$M_TRM	Terminal device

[1] Defined by the $DEVDEF macro

The first byte of the second longword returns the device class (DC$_TERM). The second byte returns the terminal type, for example, DT$_VT52. The $DCDEF macro defines the symbols for terminal class and type.

The third and fourth bytes of the second longword return the page width. The page width can have a value in the range of 1 to 511. The driver does not accept a value of 0.

The third longword contains terminal characteristics in the first three bytes and page length in the fourth byte. Terminal characteristics are normally set at system generation time to any one of, or a combination of, the values listed in Table 2-4. The $TTDEF macro defines symbols for terminal characteristics. Page length can have a value in the range of 0 to 255.

The Set Mode and Set Characteristics function (see Section 2.4.3) and the Set Terminal command are used to change terminal characteristics. The VAX/VMS Command Language User's Guide describes the Set Terminal command.

Table 2-4
Terminal Characteristics

Value[1]	Meaning
TT$M_CRFILL	Terminal requires fill after RET (the fill type can be specified by the Set Mode function P4 argument).
TT$M_EIGHTBIT	Terminal uses 8-bit ASCII character set. Terminals without this characteristic use the 7-bit ASCII code. In this case, the eighth bit is masked out on received characters and ignored on output characters. The eighth bit is meaningful only if TT$M_EIGHTBIT is set.
TT$M_ESCAPE	Terminal generates escape sequences (see Section 2.2.4). Escape sequences are validated for syntax.
TT$M_HALFDUP	Terminal is in half-duplex mode (see Section 2.2.8). All reads and writes are executed sequentially.
TT$M_HOLDSCREEN	Terminal is in Holdscreen Mode. The driver automatically causes the terminal to enter or exit from the mode when the mode is changed at the terminal. This mode is meaningful only to the DEC VT-52 and VT-55 terminals (see the DECscope User's Guide) and the VT-100 terminal (see the VT100 User Guide).
TT$M_HOSTSYNC	Host/terminal synchronization. CTRL/Q and CTRL/S are used to control data flow and thus keep the type-ahead buffer from filling.
TT$M_LFFILL	Terminal requires fill after LF (the fill can be specified by the Set Mode P4 argument).
TT$M_LOWER	Terminal has lower case character set. Unless the terminal is in the PASSALL mode or IO$M_NOFORMAT is specified, all input, output, and echoed lowercase characters (hexadecimal 61 to 7A) are converted to uppercase if TT$M_LOWER is not set.
TT$M_MBXDSABL	Mailboxes associated with the terminal will not receive unsolicited input notification or hang-up notification (see Section 2.2.7). This bit can be set by the IO$M_DSABLMBX function modifier for reads and cleared by the IO$M_ENABLMBX function modifier for writes.
TT$M_MECHFORM	Terminal has mechanical form feed. The driver passes form feeds directly to the terminal instead of expanding to line feeds.

[1] Prefix can be TT$M_ or TT$V_. TT$M_ is a bit mask whose bit corresponds to the specific field; TT$V_ is a bit number.

(continued on next page)

Table 2-4 (Cont.)
Terminal Characteristics

Value[1]	Meaning
TT$M_MECHTAB	Terminal has mechanical tabs. In order to accomplish correct line wrapping, MODULO (8) is assumed.
TT$M_NOBRDCST	Terminal will not receive any broadcast messages.
TT$M_NOECHO	Input characters are not echoed on this terminal line (see Section 2.2.1).
TT$M_NOTYPEAHD	Data must be solicited by a read operation. Data is lost if received in the absence of an outstanding read request, that is, unsolicited data. Disables type-ahead capability (see Section 2.2.1).
TT$M_PASSALL	Terminal is in PASSALL mode; all input and output data is in binary (no data interpretation occurs). Data termination occurs when the buffer is full or the read data matches the specified terminator. (See Section 2.4.1 for a comparison with the read QIO function IO$_READPBLK.)
TT$M_READSYNC	Read synchronization. The host explicitly solicits all read operations by issuing a CTRL/Q and terminates the operation by issuing a CTRL/S.
TT$M_REMOTE	Dial-up terminal. Terminal returns to local mode when a hang-up occurs on the terminal line (see Section 2.2.5). This characteristic cannot be changed; it is only informational.
TT$M_SCOPE	Terminal is a video screen display (CRT terminal), for example, the VT-52 or VT-100.
TT$M_TTSYNC	Terminal/host synchronization. Output to the terminal is controlled by terminal-generated CTRL/Q and CTRL/S.
TT$M_WRAP	A newline should be inserted if the cursor moves beyond the right margin. If TT$M_WRAP is not set, no newline is sent.

[1] Prefix can be TT$M_ or TT$V_. TT$M_ is a bit mask whose bit corresponds to the specific field; TT$V_ is a bit number.

2.4 TERMINAL FUNCTION CODES

The basic terminal I/O functions are read, write, and set mode or characteristics (see Section 1.5). All three I/O functions can take function modifiers. There are two set mode or characteristics functions: Set Mode (IO$_SETMODE) and Set Characteristic (IO$_SETCHAR).

2.4.1 Read

When a read function code is issued, the user-specified buffer is filled with characters from the associated terminal. VAX/VMS defines four basic read functions, which are listed with their function codes below:

- IO$_READVBLK - read virtual block
- IO$_READLBLK - read logical block
- IO$_READPROMPT - read with prompt
- IO$_READPBLK - read physical block
- IO$_TTYREADALL - read passall (virtual or logical block)
- IO$_TTYREADPALL - read passall with prompt (virtual or logical block)

Read operations are terminated if either of the following conditions occurs:

- The user buffer is full
- The received character is included in a specified terminator mask (see Section 2.4.1.2)

The read function codes can take all six device/function-dependent arguments (Pl through P6) on QIO requests:

- Pl = the starting virtual address of the buffer that is to receive the data read
- P2 = the size of the buffer that is to receive the data read in bytes. A system generation parameter, MAXBUF, limits the maximum size of the buffer.
- P3 = read with timeout, timeout count (see Table 2-5, IO$M_TIMED)
- P4 = the read terminator descriptor block address (see Section 2.4.1.2)
- P5 = the starting virtual address of the prompt buffer that is to be written to the terminal. For read with prompt operations (IO$_READPROMPT or IO$_TTYREADPALL).
- P6 = the size of the prompt buffer that is to be written to the terminal. For read with prompt operations (IO$_READPROMPT or IO$_TTYREADPALL).

In a read with prompt operation, the P5 and P6 arguments specify the address and size of a prompt string buffer containing data to be written to the terminal before the input data is read. In a read with prompt operation, a write operation and a read operation are performed on the specified terminal. The prompt string buffer is formatted like any other write buffer, but no carriage control can be implicitly specified. (Carriage control specifiers are described in Section 2.4.2.2.)

During a read with prompt operation, typing CTRL/O (which is turned off at the start of any read) stops the prompt string. If CTRL/U or CTRL/X is typed, the entire prompt string is written out again and the current input is ignored. If CTRL/R is typed, the current prompt string and input are written to the terminal.

Depending on the terminal type and the user's input, the prompt string can be very simple or quite complex -- from single command prompts to screen fills followed by input data.

In PASSALL mode, data received from the associated terminal is placed in the user buffer as binary information without interpretation. There are three ways to place the terminal driver in a temporary PASSALL mode for the duration of a single read QIO:

1. IO$_READPBLK -- reads a physical block without interpreting the data. Physical I/O privilege is required.

2. IO$_TTYREADALL -- allows nonprivileged users to bypass terminal driver interpretation of data.

3. IO$_TTYREADPALL -- performs the same function as IO$_TTYREADALL after writing a prompt string.

These functions are in contrast with the more comprehensive PASSALL mode established by the Set Mode characteristic TT$M_PASSALL. All input and output data is in 8-bit binary format when TT$M_PASSALL is set (see Section 2.4.3).

Since IO$_READPBLK, IO$_TTYREADALL, and IO$_TTYREADPALL do not purge the type-ahead buffer (unless requested using the IO$M_PURGE function modifier) the characters in the type-ahead buffer may have been subjected to CTRL/Y/C/S/Q/O interpretation.

2.4.1.1 **Function Modifier Codes for Read QIO Functions** - Eight function modifiers can be specified with IO$_READVBLK, IO$_READLBLK, IO$_READPROMPT, IO$_READPBLK, IO$_TTYREADALL and IO$_TTYREADPALL. Table 2-5 lists these function modifiers. IO$M_CVTLOW and IO$M_NOFILTR are not meaningful to IO$_READPBLK, IO$_TTYREADALL, and IO$_TTYREADPALL.

Table 2-5
Read QIO Function Modifiers

Code	Consequence
IO$M_CVTLOW	Lowercase alphabetic characters (hexadecimal 61 to 7A) are converted to uppercase when transferred to the user buffer or echoed. Only for IO$_READLBLK, IO$_READVBLK, and IO$_READPROMPT.
IO$M_DSABLMBX	The mailbox is disabled for unsolicited data.
IO$M_NOECHO	Characters are not echoed (that is, displayed) as they are entered at the keyboard. The terminal line can also be set to a "no echo" mode by the Set Mode characteristic TT$M_NOECHO, which inhibits all read operation echoing.

(continued on next page)

Table 2-5 (Cont.)
Read QIO Function Modifiers

Code	Consequence
IO$M_NOFILTR	The terminal does not interpret CTRL/U, CTRL/R, or DEL. They are passed to the user. Only for IO$_READLBLK, IO$_READVBLK, and IO$_READPROMPT.
IO$M_PURGE	The type-ahead buffer is purged before the read operation begins.
IO$M_REFRESH	If the read operation is interrupted by a write (either a write breakthrough or any other type of write), display the current read data when the read function is restarted.
IO$M_TIMED	The P3 argument specifies the maximum time (seconds) that can elapse between characters received; that is, the timeout value for the operation. If the read does not complete within the specified time, a timeout error (SS$_TIMEOUT) is returned. All input characters received before the read timed out are returned in the user's buffer. A read with timeout operation in which the timeout value is 0 empties the type-ahead buffer into the user buffer until the proper termination condition is reached (buffer full, termination character detected, or type-ahead buffer empty). The read operation then returns the count of characters read and, if a terminator is read, SS$_NORMAL; SS$_TIMEOUT is returned if no terminator is read. In either case the byte count in the IOSB always indicates the number of characters read.
IO$M_TRMNOECHO	The termination character (if any) is not echoed. There is no formal terminator if the buffer is filled before the terminator is typed.

2.4.1.2 **Read Function Terminators** - The P4 argument to a read QIO function either specifies the terminator set for the read function or points to the location containing the terminator set. If P4 is 0, all ASCII characters with a code in the range 0 through 31 (hexadecimal 0 through 1F) except LF, VT, FF, TAB, and BS, are terminators. (This is the VAX-11 RMS standard terminator set.)

If P4 does not equal 0, it contains the address of a quadword that either specifies a terminator character bit mask or points to a location containing that mask. The quadword has a short form and a long form, as shown in Figure 2-3. In the short form, the correspondence is between the bit number and the binary value of the character; the character is a terminator if the bit is set. For example, if bit 0 is set, NULL is a terminator; if bit 9 is set, TAB is a terminator. If a character is not specified, it is not a terminator. Since ASCII control characters are in the range 0 through 31, the short form can be used in most cases.

The long form allows use of a more comprehensive set of terminator characters. Any mask equal to or greater than 1 byte is acceptable. For example, a mask size of 16 bytes allows all 7-bit ASCII characters to be used as terminators; a mask size of 32 bytes allows all 8-bit characters to be used as terminators for 8-bit terminals.

If the terminator mask is all 0's, there are no specified terminators. The read operation ends when the specified number of characters have been transferred to the input buffer.

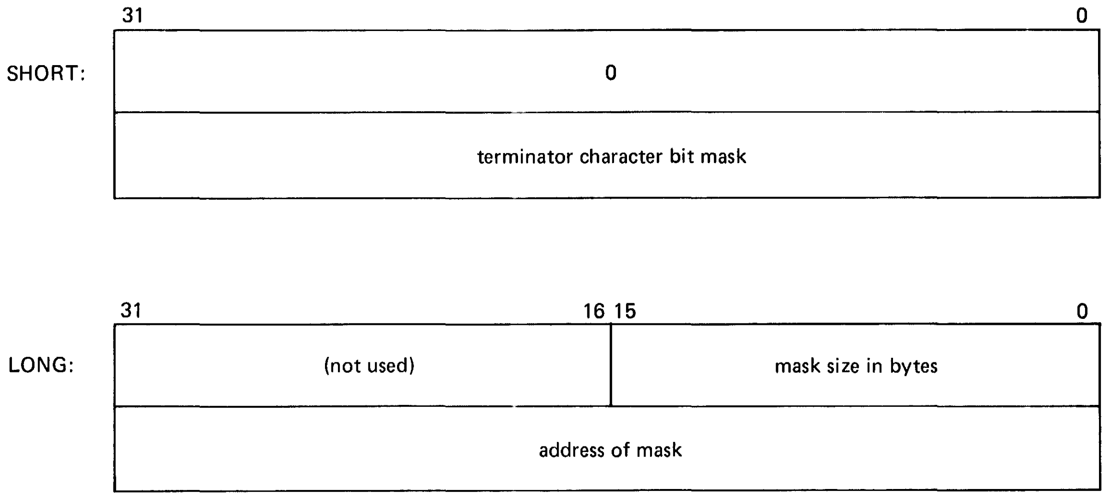


Figure 2-3 Short and Long Forms of Terminator Mask Quadwords

2.4.2 Write

Write operations display the contents of a user-specified buffer on the associated terminal. VAX/VMS defines three basic write I/O functions, which are listed with their function codes below:

- IO$_WRITEVBLK - write virtual block
- IO$_WRITELBLK - write logical block
- IO$_WRITEPBLK - write physical block

The write function codes can take the following device/function-dependent arguments:

- P1 = the starting virtual address of the buffer that is to be written to the terminal
- P2 = the number of bytes that are to be written to the terminal. A system generation parameter, MAXBUF, limits the maximum size of the buffer.
- P3 (ignored)
- P4 = carriage control specifier except for write physical block operations. (Write function carriage control is described in Section 2.4.2.2.)

P3, P5, and P6 are not meaningful for terminal write operations.

In write virtual block and write logical block operations, the buffer (P1 and P2) is formatted for the selected terminal and includes the carriage control information specified by P4.

All lowercase characters are converted to uppercase if the characteristics of the selected terminal do not include TT$M_LOWER (this does not apply to write physical block operations or when IO$M_NOFORMAT is specified).

Unless TT$M_MECHFORM is specified, multiple line feeds are generated for form feeds. The number of line feeds generated depends on the current page position and the length of the page. By producing a carriage return after the last line feed, a form feed also moves the cursor to the left margin. Multiple spaces are generated for tabs if the characteristics of the selected terminal do not include TT$M_MECHTAB (this does not apply to write physical block operations). Tab stops are every 8 characters or positions (that is, 1, 8, 16, 24,...).

2.4.2.1 **Function Modifier Codes for Write QIO Functions** - Four function modifiers can be specified with IO$_WRITEVBLK, IO$_WRITELBLK, and IO$_WRITEPBLK. Table 2-6 lists these function modifiers.

Table 2-6
Write QIO Function Modifiers

Code	Consequence
IO$M_CANCTRLO	Turns off CTRL/O (if it is in effect) before the write. Otherwise, the data may not be displayed.
IO$M_ENABLMBX	Enables use of the mailbox associated with the terminal for notification that unsolicited data is available.
IO$M_NOFORMAT	Allows nonprivileged users to write information without interpretation or format; in effect the terminal line is in a temporary PASSALL mode.
IO$M_REFRESH	If a read operation is interrupted by a write (either a write breakthrough or any other type of write), display the current read data when the read function is restarted.

2.4.2.2 **Write Function Carriage Control** - The P4 argument is a longword that specifies carriage control. Carriage control determines the next printing position on the terminal. P4 is ignored in a write physical block operation. Figure 2-4 shows the P4 longword format.

Only bytes 0, 2, and 3 in the longword are used. Byte 1 is ignored. If the low-order byte (byte 0) is not 0, the contents of the longword are interpreted as a FORTRAN carriage control specifier. Table 2-7 lists the possible byte 0 values (in hexadecimal) and their meanings.

	3	2	1	0
P4:	POSTFIX	PREFIX	(not used)	FORTRAN

Figure 2-4 P4 Carriage Control Specifier

Table 2-7
Write Function Carriage Control (FORTRAN: Byte 0 not equal to 0)

Byte Value (hexadecimal)	ASCII Character	Meaning
20	(space)	Single space carriage control. (Sequence: newline, print buffer contents, return.)
30	0	Double-space carriage control. (Sequence: newline, newline, print buffer contents, return.)
31	1	Page eject carriage control. (Sequence: form feed, print buffer contents, return.)
2B	+	Overprint carriage control. (Sequence: print buffer contents, return.) Allows double printing for emphasis or special effects.
24	$	Prompt carriage control. (Sequence: newline, print buffer contents.)
All other values		Same as ASCII space character: single-space carriage control.

If the low-order byte (byte 0) is 0, bytes 2 and 3 of the P4 longword are interpreted as the prefix and postfix carriage control specifiers. The prefix (byte 2) specifies the carriage control before the buffer contents are printed. The postfix (byte 3) specifies the carriage control after the buffer contents are printed. The sequence is:

Prefix carriage control - Print - Postfix carriage control

The prefix and postfix bytes, although interpreted separately, use the same encoding scheme. Table 2-8 shows this encoding scheme in hexadecimal.

Table 2-8
Write Function Carriage Control (P4 byte 0 = 0)

Prefix/Postfix Bytes (Hexadecimal)				
Bit 7		Bits 0 - 6		Meaning
0		0		No carriage control is specified, that is, NULL.
0		1 - 7F		Bits 0 through 6 are a count of newlines (carriage return followed by a line feed).
Bit 7	Bit 6	Bit 5	Bits 0-4	Meaning
1	0	0	1-1F	Output the single ASCII control character specified by the configuration of bits 0 through 4 (7-bit character set).
1	1	0	1-1F	Output the single ASCII control character specified by the configuration of bits 0 through 4 which are translated as ASCII characters 128 through 159 (8-bit character set).

Figure 2-5 shows the prefix and postfix hexadecimal coding that produces the carriage control functions listed in Table 2-7. Prefix and postfix coding provides an alternative way to achieve these controls.

(Space)

P4:	8D	1	–	0

Sequence:
Prefix = NL
Print
Postfix = CR

"0"

P4:	8D	2	–	0

Sequence:
Prefix = LF, LF
Print
Postfix = CR

"1"

P4:	8D	8C	–	0

Sequence:
Prefix = FF
Print
Postfix = CR

"+"

P4:	8D	0	–	0

Sequence:
Prefix = NULL
Print
Postfix = CR

"$"

P4:	0	8A	–	0

Sequence:
Prefix = NL
Print
Postfix = NULL

Example: Skip 24 lines before printing

P4:	8D	18	–	0

Sequence:
Prefix = 24NL
Print
Postfix:= CR

Figure 2-5 Write Function Carriage Control
(Prefix and Postfix Coding)

In the first example, the prefix/postfix hexadecimal coding for a single-space carriage control (newline, print buffer contents, return) is obtained by placing the value 1 in the second (prefix) byte and the sum of the bit 7 value (80) and the return value (D) in the third postfix byte:

```
  80  (bit 7 = 1)
+ D   (return)
 ---
  8D  (postfix = return)
```

2.4.3 Set Mode

Set Mode operations affect the operation and characteristics of the associated terminal line. VAX/VMS defines two types of set mode functions:

- Set Mode
- Set Characteristics

The Set Mode function affects the mode and temporary characteristics of the associated terminal line. Set Mode is a logical I/O function and requires no privilege. A single function code is provided:

- IO$_SETMODE

The Set Characteristics affects the permanent characteristics of the associated terminal line. Set Characteristics is a physical I/O function and requires the privilege necessary to perform physical I/O. A single function code is provided:

- IO$_SETCHAR

These functions take the following device/function-dependent arguments if no function modifiers are specified:

- P1 = address of characteristics buffer
- P3 = speed specifier (bits 0 through 15 only)
- P4 = fill specifier (bits 0 through 7 = CR fill count; bits 8 through 15 = LF fill count)
- P5 = parity flags

The P1 argument points to a quadword block, as shown in Figure 2-6. With the exception of terminal characteristics, the contents of the block are the same for both Set Mode and Set Characteristic functions.

The class portion of the block contains DC$_TERM, which is defined by the $DCDEF macro. Type values are defined by the $DCDEF macro, for example, DT$_LA36. Page width can have a value in the range of 1 to 511. Page length can have a value in the range of 0 to 255. Table 2-4 lists the values for terminal characteristics. These values are defined by the $TTDEF macro.

The P3 argument defines the device speed, for example, TT$C_BAUD_300. If P3 is 0, the baud rate is not changed. P4 contains fill counts for the carriage return and line feed characters. Bits 0 through 7 specify the number of fill characters used after a return. Bits 8 through 15 specify the number of fill characters used after a line feed.

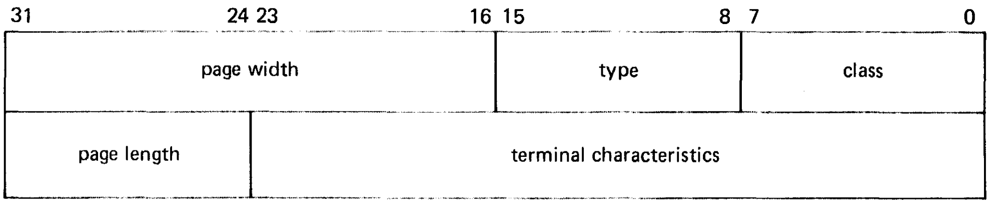


Figure 2-6 Set Mode Characteristic Buffer

(P4 is applicable only if TT$M_CRFILL or TT$M_LFFILL is specified as a terminal characteristic for the current QIO request; see Table 2-4.)

Three parity flags can be specified in the P5 argument:

TT$M_ALTRPAR - alter parity, change parity on terminal line if set

TT$M_PARITY - enable parity on terminal line if set, disable if clear

TT$M_ODD - parity is odd if set

If parity is enabled, the DZ-11 generates a parity check bit to detect parity mismatch. Parity errors that occur during an I/O read operation are fatal to the operation. Parity errors that occur when no I/O operation is in progress may result in a character loss.

The Set Mode and Set Characteristic functions can take the Enable CTRL/C AST, Enable CTRL/Y AST, and Hang-up function modifiers that are decribed below.

2.4.3.1 **Hang-Up Function Modifier** - The Hang-Up function disconnects a terminal that is on a dial-up line. (Dial-up lines are described in Section 2.2.7.) Two combinations of function code and modifier are provided:

- IO$_SETMODE!IO$M_HANGUP
- IO$_SETCHAR!IO$M_HANGUP

The Hang-up function modifier takes no arguments. SS$_NORMAL is returned in the I/O status block.

2.4.3.2 **Enable CTRL/C AST and Enable CTRL/Y AST Function Modifiers** - Both set mode functions can take the Enable CTRL/C AST and Enable CTRL/Y AST function modifiers. These function modifiers request the terminal driver to queue an AST for the requesting process when the user types CTRL/C or CTRL/Y. Enable CTRL/Y AST requires the caller to have either supervisor, executive, or kernel access mode, or logical or physical I/O privilege. Four combinations of function code and modifier are provided:

- IO$_SETMODE!IO$M_CTRLCAST - Enable CTRL/C AST
- IO$_SETMODE!IO$M_CTRLYAST - Enable CTRL/Y AST
- IO$_SETCHAR!IO$M_CTRLCAST - Enable CTRL/C AST
- IO$_SETCHAR!IO$M_CTRLYAST - Enable CTRL/Y AST

These function code modifier pairs take the following device/function-dependent arguments:

- P1 = address of the AST service or 0 if the corresponding AST is disabled
- P2 = AST parameter
- P3 = access mode to deliver AST (maximized with caller's access mode)

If the respective enable is in effect, typing CTRL/C or CTRL/Y gains the attention of the enabling process (see Table 2-1).

Enable CTRL/C and CTRL/Y AST are single (one-time) enables. After the AST occurs, it must be explicitly re-enabled by one of the four function code combinations described above before an AST can occur again. This function code is also used to disable the AST. The function is subject to AST quotas.

The user can have more than one CTRL/C or CTRL/Y enabled. All ASTs are given in their order of request, that is, first in first out, when the character is typed. For example, typing CTRL/C results in the delivery of all CTRL/C ASTs.

If no CTRL/C enable is present, the holder of a CTRL/Y enable will receive an AST when CTRL/C is typed; newline, ^Y, return is echoed.

CTRL/C enables are flushed by the Cancel I/O on Channel ($CANCEL) system service. CTRL/Y enables are flushed only during unit run down, that is, after the last deassignment by the Deassign I/O Channel ($DASSGN) system service.

CTRL/Y is normally used to gain the attention of the command interpreter and thus allow the user to input special commands such as DEBUG, STOP, CONTINUE, and so on. Thus it is recommended that programs run from a command interpreter not enable CTRL/Y. Also, since ASTs are delivered on a first-in first-out basis, the command interpreter's AST routine will get control first and possibly not allow the program's AST to be delivered at all.

Section 2.2.6 describes other effects of CTRL/C and CTRL/Y.

2.4.4 Sense Mode

The Sense Mode functions sense the characteristics of the terminal and return them to the caller in the I/O status block. Two function codes are provided:

- IO$_SENSEMODE
- IO$_SENSECHAR

IO$_SENSEMODE returns the process-associated, that is, temporary, characteristics of the terminal and IO$_SENSECHAR returns the permanent characteristics of the terminal. IO$_SENSEMODE is a logical I/O function and requires no privilege. IO$_SENSECHAR is physical I/O function and requires the privilege necessary to perform physical I/O.

These function codes take the following device/function-dependent argument:

P1 = address of a quadword characteristics buffer

The P1 argument points to a quadword block, as shown in Figure 2-7.

31 16	15 8	7 0
buffer size	type	class
terminal characteristics		

Figure 2-7 Sense Mode Characteristics Buffer

The class portion of the block contains DC$_TERM, which is defined by the $DCDEF macro. Type values are defined by the $DCDEF macro, for example, DT$_LA36. Table 2-4 lists the values for terminal characteristics. These values are defined by the $TTDEF macro.

The Sense Mode and Sense Characteristic functions can take the type-ahead count function modifier IO$M_TYPEAHDCNT.

IO$M_TYPEAHDCNT returns the count of characters presently in the type-ahead buffer and a copy of the first character in the buffer. In this case, the P1 argument points to a characteristics buffer returned by IO$M_TYPEAHDCNT. Figure 2-8 shows the format of this buffer.

31 24	23 16	15 0
(reserved)	first character	number of characters in type-ahead buffer
(reserved)		

Figure 2-8 Sense Mode Characteristics Buffer (Type-ahead)

2.5 I/O STATUS BLOCK

The I/O status block formats for the read, write, set mode, and sense mode I/O functions are shown in Figures 2-9, 2-10, and 2-11. Table 2-9 lists the status returns for these functions.

+2	IOSB
offset to terminator	status
terminator size	terminator
+6	+4

Figure 2-9 IOSB Contents - Read Function

In Figure 2-9, the offset to terminator at IOSB+2 is the count of characters before the terminator character (see Section 2.4.1.2). The terminator character(s) is in the buffer at the offset specified in IOSB+2. When the buffer is full, the offset at IOSB+2 is equal to the requested buffer size. At the same time, IOSB+4 is equal to 0. IOSB+6 contains the size of the terminator string, usually 1. However, in an escape sequence, IOSB+6 contains the size of the validated escape sequence (see Section 2.2.4). The sum of IOSB+2 and IOSB+6 is the number of characters in the buffer.

31 24	23 16	15 0
byte count		status
line position	column position	Number of lines output for the I/O function*

*0 if IO$_WRITEPBLK, IO$M_NOFORMAT, or PASSALL mode

Figure 2-10 IOSB Contents - Write Function

In Figure 2-10, the line and column positions are the terminal driver's internal computation of the cursor position after the write function has completed. Note that the terminal driver does not presently track any effects that escape sequences may have on the cursor position.

0	speed	status	
0	parity flags	LF fill count	CR fill count

Figure 2-11 IOSB Contents - Set Mode, Set Characteristics, Sense Mode, and Sense Characteristics Functions

Table 2-9
Terminal QIO Status Returns

Status	Meaning
SS$_ABORT	The operation was canceled by the Cancel I/O on Channel ($CANCEL) system service. Applicable only if the driver was actively involved in a terminal operation.
SS$_BADESCAPE	Invalid escape sequence terminator begins at the offset (IOSB+2).
SS$_CONTROLC	Read or write operation not completed because CTRL/C was typed.
SS$_CONTROLO	Write operation not completed because CTRL/O was typed.

(continued on next page)

Table 2-9 (Cont.)
Terminal QIO Status Returns

Status	Meaning
SS$_CONTROLY	Read or Write operation not completed because CTRL/Y was typed.
SS$_NORMAL	Successful completion. The operation specified in the QIO was completed successfully. On a read or write operation, the second word of the IOSB can be examined to determine the number of bytes processed. The input or output buffer contains these bytes.
SS$_PARITY	Parity bit mismatch detected by the device interface during a read operation. The I/O operation stopped when the mismatch was detected. (Data was received up to this point in the operation.) SS$_PARITY is meaningful only on terminal lines that have parity enabled.
SS$_PARTESCAPE	Partial escape sequence was stored. An escape sequence was started but read-buffer space was exhausted before the sequence was completed. The remainder of the sequence is available from the type-ahead buffer on the next read unless the terminal line has the TT$M_NOTYPEAHD characteristic (see Section 2.2.4).
SS$_TIMEOUT	Operation timeout. The specified terminal could not perform the QIO read operation because a timeout occurred at the terminal, that is, an interrupt was lost, or IO$M_TIMED was specified on a read operation (see Table 2-5), or a hardware timeout occurred. IOSB+2 contains the number of bytes transferred before the timeout occurred.

2.6 PROGRAMMING EXAMPLE

The following program shows examples of several I/O operations, using the full duplex capabilities of the driver. The program illustrates some important concepts concerning terminal driver programming: assigning an I/O channel, performing full-duplex I/O operations, and enabling CTRL/C ASTs.

The program is designed to run with a terminal set to full-duplex mode. The initialization code queues a read to the terminal and enables CTRL/C ASTs. The main loop then prints out a random message every three seconds. When the user types a message on the terminal, the read AST routine prints an acknowledgement message and queues another read. If the user types CTRL/C, the associated AST routine cancels the I/O operation on the assigned channel and exits to the command interpreter.

```
; ********************************************************************
;
;                           TERMINAL PROGRAM
;
; ********************************************************************

        .TITLE  FULL_DUPLEX TERMINAL PROGRAMMING EXAMPLE
        .IDENT  /02/

;
; DEFINE SYMBOLS
;

        $IODEF                    ; DEFINE I/O FUNCTION CODES

;
; ALLOCATE TERMINAL DESCRIPTOR AND CHANNEL NUMBER STORAGE
;

TT_DESC:
        .ASCID  /SYS$INPUT/       ; LOGICAL NAME OF TERMINAL
DEV_DESC:                         ; TRANSLATED PHYSICAL DEVICE DESCRIPTOR
PHYS_NAME_LEN:
        .LONG   63
PHYS_NAME_ADR:
        .LONG   PHYS_NAME
PHYS_NAME:
        .BLKB   63
TT_CHAN:
        .BLKW   1                 ; TT CHANNEL NUMBER STORAGE

;
; ALLOCATE INPUT BUFFER
;

IN_BUF:
        .BLKB   20                ; 20 CHARACTER BUFFER
IN_BUFLEN=.-IN_BUF                ; CALCULATE LENGTH OF BUFFER
IN_IOSB:
        .BLKQ   1                 ; INPUT I/O STATUS BLOCK

;
; DEFINE CARRIAGE CONTROL SYMBOLS
;

        CR=^X0D                   ; CARRIAGE RETURN
        LF=^X0A                   ; LINE FEED

;
; DEFINE ACKNOWLEDGEMENT MESSAGE
;

ACK_MSG:
        .ASCII  /INPUT ACKNOWLEDGED/<CR><LF>
ACK_MSGLEN=.-ACK_MSG              ; CALCULATE LENGTH OF MESSAGE

;
; DEFINE OUTPUT MESSAGES
;
; OUTPUT MESSAGES ARE ACCESSED BY INDEXING INTO A TABLE OF
; LONGWORDS WITH EACH MESSAGE DESCRIBED BY A MESSAGE ADDRESS AND
; MESSAGE LENGTH
;
```

```
ARRAY:                              ; TABLE OF MESSAGE ADDRESSES AND
                                    ; LENGTHS
        .LONG   10$                 ; FIRST MESSAGE ADDRESS
        .LONG   15$                 ; FIRST MESSAGE LENGTH
        .LONG   20$
        .LONG   25$
        .LONG   30$
        .LONG   35$
        .LONG   40$
        .LONG   45$

;
; DEFINE MESSAGES
;

10$:    .ASCII  /RED ALERT!!!   RED ALERT!!!/<CR><LF>
15$=.-10$
;
20$:    .ASCII  /ALL SYSTEMS GO/<CR><LF>
25$=.-20$
;
30$:    .ASCII  /WARNING.....INTRUDER ALARM/<CR><LF>
35$=.-30$
;
40$:    .ASCII  /***** SYSTEM OVERLOAD *****/<CR><LF>
45$=.-40$

;
; STATIC QIO PACKET FOR MESSAGE OUTPUT USING QIO$_G FORM
;

WRITE_QIO:
        $QIO    FUNC=IO$_WRITEVBLK,EFN=1        ; QIO PACKET

;
; TIMER STORAGE
;

WAITIME:
        .LONG   -10*1000*1000*3,-1      ; 3 SECOND DELTA TIME
TIME:
        .BLKQ   1                   ; CURRENT STORAGE TIME USED FOR
                                    ; RANDOM NUMBER
;
;
; *******************************************************************
;
;                     START PROGRAM
;
; *******************************************************************
;
; THE FOLLOWING CODE PERFORMS INITIALIZATION FUNCTIONS.  THE PROGRAM
; ASSUMES THAT THE TERMINAL IS ALREADY IN FULL-DUPLEX MODE.
;

START:
        .WORD   0                   ; ENTRY MASK
        $TRNLOG_S       -           ; GET TERMINAL'S PHYSICAL DEVICE NAME
                LOGNAM=TT_DESC,-
                RSLLEN=PHYS_NAME_LEN,-
                RLSBUF=DEV_DESC
        CMPB    PHYS_NAME,#^X1B ; DOES NAME START WITH ESCAPE ?
        BNEQ    5$                  ; NO
        SUBL    #4,PHYS_NAME_LEN        ; YES, STRIP OFF FIRST 4 CHARS
        ADDL    #4,PHYS_NAME_ADR
;
5$:
        $ASSIGN_S       DEVNAM=DEV_DESC,CHAN=TT_CHAN    ; ASSIGN
                                        ; TERMINAL CHANNEL
        BLBS    R0,10$                  ; NO ERROR IF SET
        BRW     ERROR                   ; ERROR
10$:    BSBW    ENABLE_CTRLCAST         ; ALLOW CONTROL/C TRAPS
        BSBW    ENABLE_READ             ; QUEUE READ
        MOVZWL  TT_CHAN,WRITE_QIO+8     ; INSERT CHANNEL INTO STATIC
                                        ; QIO PACKET

;
; THIS LOOP OUTPUTS A MESSAGE BASED ON A RANDOM NUMBER AND THEN DELAYS
; FOR 3 SECONDS
;

LOOP:
        $GETTIM_S       TIMADR=TIME     ; GET RANDOM TIME
        BLBS    R0,10$                  ; NO ERROR IF SET
        BRW     ERROR
10$:    EXTZV   #6,#2,TIME,R0           ; LOAD RANDOM BITS INTO SWITCH
        MOVQ    ARRAY[R0],WRITE_QIO+28  ; LOAD MESSAGE ADDRESS
                                        ; AND SIZE INTO QIO PACKET

;
; ISSUE QIO WRITE USING PACKET DEFINED IN DATA AREA
;

        $QIOW_G WRITE_QIO
        BLBS    R0,5$                   ; NO ERROR IF SET
        BRW     ERROR

;
; DELAY FOR 3 SECONDS BEFORE ISSUING NEXT MESSAGE
;

5$:
        $SETIMR_S  EFN=#2,DAYTIM=WAITIME        ; TIMER SERVICE WILL
                                        ; SET EVENT FLAG IN 3 SECONDS
        BLBS    R0,20$                  ; NO ERROR IF SET
        BRW     ERROR
20$:    $WAITFR_S       EFN=#2          ; WAIT FOR EVENT FLAG
        BLBS    R0,LOOP                 ; NO ERROR IF SET
        BRW     ERROR

;
; ROUTINE TO ALLOW CONTROL C RECOGNITION
;

ENABLE_CTRLCAST:
        $QIOW_S CHAN=TT_CHAN,-
                FUNC=#IO$_SETMODE!IO$M_CTRLCAST,-
                P1=CTRLCAST,-           ; AST ROUTINE ADDRESS
                P3=#3                   ; USER MODE
        BLBS    R0,10$                  ; NO ERROR IF SET
        BRW     ERROR
10$:    RSB

;
; AST ROUTINE TO EXECUTE WHEN CONTROL C IS TYPED
;

CTRLCAST:
        .WORD    ^M<R2,R3,R4,R5>           ; PROCEDURE ENTRY MASK
        $CANCEL_S       CHAN=TT_CHAN       ; FLUSH ANY I/O ON QUEUE

; WHEN USING FULL-DUPLEX, ASYNCHRONOUS I/O, THE USER MUST ISSUE CANCEL
; ON ANY OUTSTANDING I/O.  THIS MUST BE DONE TO PREVENT ANY
; OUTSTANDING QUEUED READS FROM BLOCKING DCL'S PROMPT MESSAGE.  DCL
; PERFORMS ITS OWN CANCEL ON ITS OWN CHANNEL, NOT ONE DEFINED BY THE
; USER.

        $EXIT_S                            ; EXIT TO DCL

;
; ROUTINE TO QUEUE A READ TO THE TERMINAL
;

ENABLE_READ:
        $QIO_S  CHAN=TT_CHAN,-             ; MUST NOT BE QIOW FORM
                                           ; OR READ WILL BLOCK PROCESS
                FUNC=#IO$_READVBLK,-
                IOSB=IN_IOSB,-
                ASTADR=READAST,-           ; AST ROUTINE TO EXECUTE ON
                P1=IN_BUF,-
                P2=#IN_BUFLEN
        BLBS    R0,10$                     ; NO ERROR IF SET
        BRW     ERROR

; THE QUEUED READ WILL NOT AFFECT WRITES UNTIL THE FIRST CHARACTER IS
; TYPED.  IF NO WRITES ARE ACTIVE AT THAT TIME, THE READ BECOMES
; CURRENT AND SUBSEQUENT WRITES ARE BLOCKED UNTIL THE READ COMPLETES.
; IF WRITES ARE ACTIVE, TYPED CHARACTERS ARE STORED IN THE TYPE AHEAD
; BUFFER UNTIL THE WRITE QUEUE EMPTIES.

10$:    RSB

;
; AST ROUTINE TO EXECUTE ON READ COMPLETION
;

READAST:
        .WORD   ^M<R2,R3,R4,R5>            ; PROCEDURE ENTRY MASK
        $QIO_S  CHAN=TT_CHAN,-             ; ISSUE ACKNOWLEDGE MESSAGE
                FUNC=#IO_WRITEVBLK,-
                P1=ACK_MSG,-
                P2=#ACK_MSGLEN

;
; PROCESS READ MESSAGE
;
        .
        .
        .
(User provided code to decode command inserted here)
        .
        .
        .

        BSBW    ENABLE_READ                ; QUEUE NEXT READ
        RET                                ; RETURN TO MAINLINE LOOP

; ERROR ROUTINE
;

ERROR:
```

```
$EXIT_S R0                               ; EXIT WITH STATUS ERROR
                                         ; RETURN
.END    START
```

CHAPTER 3

DISK DRIVERS

This chapter describes the use of the VAX/VMS disk drivers. These drivers support the devices listed in Table 3-1 and detailed in Section 3.1.

Table 3-1
Disk Devices

Model	Type[1]	RPM	Surfaces	Cylinders	Bytes/ Track	Bytes/ Drive
RM03	Pack	3600	5	823	16,384	67,420,160
RP05	Pack	3600	19	411	11,264	87,960,576
RP06	Pack	3600	19	815	11,264	174,423,040
RK06	Cart	2400	3	411	11,264	13,888,512
RK07	Cart	2400	3	815	11,264	27,550,480
RX01	Flex	360	1	77	3,328	256,256

1. Pack = pack disk; Cart = cartridge disk; Flex = flexible diskette (floppy)

All disk drivers support Files-11 Structure Level 1 and Level 2 file structures. Access to these file structures is through the standard MOUNT and INIT DCL commands followed by the RMS-32 calls described in the VAX-11 Record Management Services Manual. Files in RT-11 format can be read or written with the file exchange facility FLX.

The contents of disk bootstrap blocks are CPU- and operating system-dependent. For the LSI-11 Console on the VAX-11/780, the standard bootstrap for the RT-11 operating system is used. Your software support specialist can provide more information on the RT-11 bootstrap.

3.1 SUPPORTED DISK DEVICES

The following sections provide greater detail on each of the disk devices listed in Table 3-1.

3.1.1 RM03 Pack Disk

The RM03 pack disk is a removable, moving head disk that consists of 5 data surfaces. The RM03 is connected to the system by a MASSBUS adapter (MBA). Up to eight drives can be connected to each MBA.

3.1.2 RP05 and RP06 Pack Disks

The RP05 and RP06 pack disks consist of 19 data surfaces and a moving read/write head. The RP06 pack disk has approximately twice the capacity of the RP05. These disks are connected to the system by an MBA. Up to eight drives can be connected to each MBA.

3.1.3 RK06 and RK07 Cartridge Disks

The RK06 cartridge disk is a removable, random-access, bulk-storage device with three data surfaces. The RK07 cartridge disk is a double-density RK06. The RK06 and RK07 are connected to the system by an RK611 controller which interfaces to the UNIBUS adapter (UBA). Up to eight disk drives can be connected to each RK611.

3.1.4 RX01 Console Disk

The RX01 floppy disk uses a flexible "diskette" or "floppy" disk. The disk is connected to the LSI console on the VAX-11/780, which the driver accesses using the MTPR and MFPR privileged instructions.

For read or write physical block operations, the track, sector, and cylinder parameters shown in Figure 3-2 describe a physical, 128-byte RX01 sector. Note that the driver does not apply track-to-track skew, cylinder offset, or sector interleaving to this physical media address. Sector numbers are interleaved to expedite data transfers. Section 3.2.4 describes this feature in greater detail.

3.2 DRIVER FEATURES AND CAPABILITIES

The VAX/VMS disk drivers provide the following capabilities:

- Multiple controllers of the same type; for example, more than one MBA or RK611 can be used on the system
- Up to eight drives per controller (depending on the device)
- Different types of drive on a single controller (MBA only)
- Overlapped seeks (except RX01)
- Data checks on a per-request, per-file, and/or per-volume basis (except RX01)
- Full recovery from power failure for online drives with volumes mounted
- Extensive error recovery algorithms; for example, error code correction and offset (except RX01)

- Dynamic, as well as static, bad block handling
- Logging of device errors in a file that can be displayed by field service personnel or customer personnel
- Online diagnostic support for drive level diagnostics
- Multiple block, noncontiguous, virtual I/O operations at the driver level
- Optimization of physical sector translation (RX01 only)

The following sections describe the data check, overlapped seek, error recovery, and logical to physical translation capabilities in greater detail.

3.2.1 Data Check

A data check is made after successful completion of a read or write operation and compares the data in memory with the data on disk to make sure they match.

Disk drivers support data checks at three levels:

- Per request -- Users can specify the data check function modifier (IO$M_DATACHECK) on a read logical block, write logical block, read virtual block, write virtual block, read physical block, or write physical block operation.
- Per volume -- Users can specify the characteristics "data check all reads" and/or "data check all writes" when the volume is mounted. The VAX/VMS Command Language User's Guide describes volume mounting and dismounting.
- Per file -- Users can specify the file access attributes "data check on read" or "data check on write." File access attributes are specified when the file is accessed. Chapter 9 of this manual and the VAX-11 Record Management Services Reference Manual describe file access.

Offset recovery is performed during a data check but Error Code Correctable (ECC) correction is not (see Section 3.2.3). This means that if a read operation is performed and an ECC correction applied, the data check would fail even though the data in memory is correct. In this case, the driver returns a status code indicating that the operation was successfully completed, but that the data check could not be performed because of an ECC correction.

Data checks on read operations are extremely rare and users can either accept the data as is, treat the ECC correction as an error, or accept the data but immediately move it to another area on the disk volume.

3.2.2 Overlapped Seeks

A seek operation involves moving the disk read/write heads to a specific place on the disk without any transfer of data. All transfer functions, including data checks, are preceded by an implicit seek operation (except when the seek is inhibited by the physical I/O function modifier IO$M_INHSEEK). Except on RX01 drives, seek operations can be overlapped. That is, when one drive performs a seek operation, any number of other drives can also perform seek operations.

During the seek operation, the controller is free to perform transfers on other units. Thus, seek operations can also overlap data transfer operations. For example, at any one time, seven seeks and one data transfer could be in progress on a single controller.

This overlapping is possible because, unlike I/O transfers, seek operations do not require the controller once they are initiated. Therefore, seeks are initiated before I/O transfers and other functions that require the controller for extended periods.

3.2.3 Error Recovery

Error recovery in VAX/VMS is aimed at performing all possible operations to successfully complete an I/O operation. Error recovery operations fall into four categories:

- Handling special conditions such as power failure and interrupt timeout
- Retrying nonfatal controller and/or drive errors
- Applying error correction information (not applicable for RX01)
- Offsetting read heads to try to obtain a stronger recorded signal (not applicable for RX01)

The error recovery algorithm uses a combination of these four types of error recovery operations to complete an I/O operation.

Power failure recovery consists of waiting for mounted drives to spin up and come on line followed by reexecution of the I/O operation that was in progress at the time of the power failure.

Device timeout is treated as a nonfatal error. The operation that was in progress when the timeout occurred is reexecuted up to eight times before a timeout error is returned.

Nonfatal controller/drive errors are simply reexecuted up to eight times before a fatal error is returned.

All normal error recovery (nonspecial conditions) can be inhibited by specifying the inhibit retry function modifier (IO$M_INHRETRY). If any error occurs and this modifier is specified, the virtual, logical, or physical I/O operation is immediately terminated, and a failure status is returned. This modifier has no effect on power recovery and timeout recovery.

3.2.4 Logical to Physical Translation (RX01)

Logical block to physical sector translation on RX01 drives adheres to the standard VAX/VMS format. For each 512-byte logical block selected, the driver reads or writes four 128-byte physical sectors. To minimize rotational latency, the physical sectors are interleaved. This allows the processor time to complete a sector transfer before the next sector in the block reaches the read/write heads. To allow for track to track switch time, the next logical sector that falls on a new track is skewed by six sectors. (There is no interleaving or skewing on read physical block and write physical block I/O operations.) Logical blocks are allocated starting at track 1; track 0 is not used.

The translation procedure, in more precise terms, is as follows:

1. Compute an uncorrected media address using the following dimensions:

   Number of sectors per track = 26

   Number of tracks per cylinder = 1

   Number of cylinders per disk = 77

2. Correct the computed address for interleaving and track-to-track skew (in that order) as shown in the following VAX-11 FORTRAN statements. ISECT is the sector address and ICYL is the cylinder address computed in step 1:

   Interleaving:

   ```
   ITEMP = ISECT*2
   IF (ISECT .GT. 12) ITEMP = ITEMP+1
   ISECT = ITEMP
   ```

   Skew:

   ```
   ISECT = ISECT+(6*ICYL)
   ISECT = MOD (ISECT, 26)
   ```

3. Set the sector number in the range 1 through 26 as required by the hardware:

   ```
   ISECT = ISECT+1
   ```

4. Adjust the cylinder number to cylinder 1 (cylinder 0 is not used):

   ```
   ICYL = ICYL+1
   ```

3.3 DEVICE INFORMATION

Users can obtain information on all disk device characteristics by using the $GETCHN and $GETDEV system services (see Section 1.10). The disk-specific information is returned in the first three longwords and in the last longword of a user-specified buffer, as shown in Figure 3-1 (Figure 1-9 shows the entire buffer).

Table 3-2 lists the device characteristics returned in the first longword.

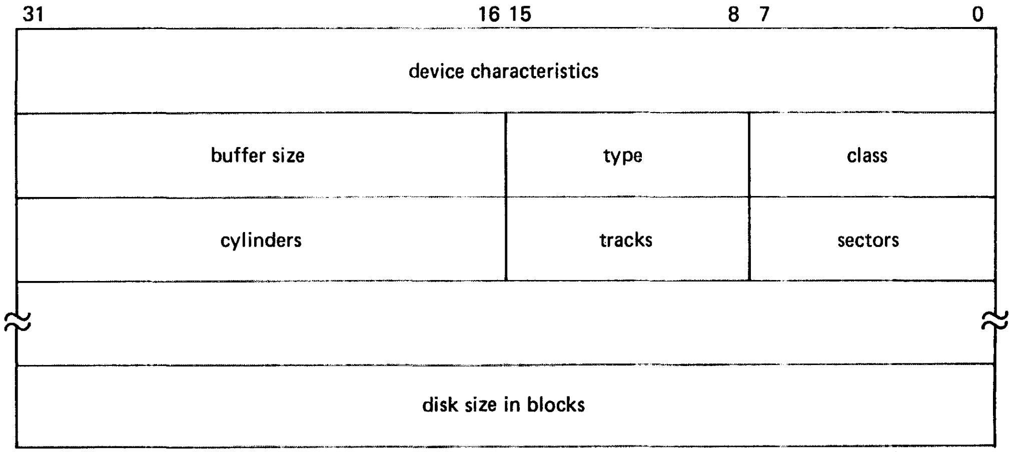


Figure 3-1 Disk Information

Table 3-2
Disk Device Characteristics

Dynamic Bits[1] (Conditionally Set)	Meaning
DEV$M_AVL	Device is on line and available
DEV$M_FOR	Foreign device
DEV$M_MNT	Volume mounted
DEV$M_RCK	Perform data check all reads
DEV$M_WCK	Perform data check all writes
Static Bits[1] (Always Set)	**Meaning**
DEV$M_FOD	File-oriented device
DEV$M_IDV	Device is capable of input
DEV$M_ODV	Device is capable of output
DEV$M_RND	Device is capable of random access
DEV$M_SHR	Device is shareable

[1] Defined by the $DEVDEF macro.

The second longword contains information on the device class and type, and the buffer size. The device class for disks is DC$_DISK and the device types are:

Device Type	Disk
DT$_RM03	RM03
DT$_RP05	RP05
DT$_RP06	RP06
DT$_RK06	RK06
DT$_RK07	RK07
DT$_RX01	RX01

The $DCDEF macro defines the device type and class names. The buffer size is the default to be used for disk transfers (this default is normally 512 bytes).

The third longword contains information on the number of cylinders per disk, the number of tracks per cylinder, and the number of sectors per track.

The last longword contains the maximum number of blocks (1 block = 512 bytes) that can be contained on the disk.

3.4 DISK FUNCTION CODES

VAX/VMS disk drivers can perform logical, virtual, and physical I/O functions.

Logical and physical I/O functions allow access to volume storage and require only that the issuing process have access to the volume. Virtual I/O functions require intervention by an Ancillary Control Process (ACP) and must be executed in a prescribed order. The normal procedure is to create a file and access it. Information is then written to the file and the file is deaccessed. The file is subsequently accessed, the information is read, and the file is deaccessed. The file is deleted when the information it contains is no longer useful.

Any number of blocks (up to a maximum of 64K bytes) can be read or written by a single request. The number itself has no effect on the applicable quotas (direct I/O, buffered I/O, and AST). Reading or writing 1 block or 10 blocks subtracts the same amount from the quota.

The volume to which a logical or virtual function is directed must be mounted in order for the function to actually be executed. If it is not mounted, either a "device not mounted" or "invalid volume" status is returned in the I/O status block.

Table 3-3 lists the logical, virtual, and physical disk I/O functions and their function codes. Chapter 9 describes the QIO level interface to the disk device ACP.

Table 3-3
Disk I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_CREATE P1,[P2],- [P3],[P4],[P5]	V	IO$M_CREATE IO$M_ACCESS IO$M_DELETE	Create a directory entry or a file
IO$_ACCESS P1, [P2],- [P3],[P4],[P5]	V	IO$M_CREATE IO$M_ACCESS	Search a directory for a specified file and access the file if found
IO$_DEACCESS P1,[P2],- [P3],[P4],[P5]	V		Deaccess a file and if specified, write final attributes in the file header
IO$_MODIFY P1,[P2], [P3],[P4],[P5]	V		Modify the file attributes and/or allocation
IO$_DELETE P1,[P2],- [P3],[P4],[P5]	V	IO$M_DELETE	Remove a directory entry and/or file header
IO$_ACPCONTROL P1,- [P2],[P3],[P4],[P5]	V	IO$M_DMOUNT	Perform miscellaneous control functions (see Section 9.3)
IO$_MOUNT	V		Informs ACP when volume is mounted; requires mount privilege
IO$_READVBLK P1,P2,P3	V	IO$M_DATACHECK[2] IO$M_INHRETRY	Read virtual block
IO$_READLBLK P1,P2,P3	L	IO$M_DATACHECK[2] IO$M_INHRETRY	Read logical block
IO$_READPBLK P1,P2,P3	P	IO$M_DATACHECK[2] IO$M_INHRETRY IO$M_INHSEEK[2]	Read physical block
IO$_WRITEVBLK P1,P2,P3	V	IO$M_DATACHECK[2] IO$M_INHRETRY	Write virtual block
IO$_WRITELBLK P1,P2,P3	L	IO$M_DATACHECK[2] IO$M_INHRETRY	Write logical block

[1] V = virtual; L = logical; P = physical

[2] Except for RX01

(continued on next page)

Table 3-3 (Cont.)
Disk I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_WRITEPBLK P1,P2,P3	P	IO$M_DATACHECK[2] IO$M_INHRETRY IO$M_INHSEEK[2]	Write physical block
IO$_SETMODE P1	L		Set disk characteristics for subsequent operations
IO$_SETCHAR P1	P		Set disk characteristics for subsequent operations
IO$_SENSEMODE	L		Sense the device-dependent characteristics and return them in the I/O status block
IO$_SENSECHAR	P		Sense the device-dependent characteristics and return them in the I/O status block
IO$_SEARCH P1	P		Search for specified block or sector
IO$_PACKACK	P		Initialize volume valid
IO$_SEEK P1	P		Seek to specified cylinder
IO$_WRITECHECK P1,-P2,P3	P		Verify data written to disk by a previous write QIO

[1] V = virtual; L = logical; P = physical

[2] Except for RX01

The function-dependent arguments for IO$_CREATE, IO$_ACCESS, IO$_DEACCESS, IO$_MODIFY, and IO$_DELETE are:

- P1 -- the address of the File Information Block (FIB) descriptor.
- P2 -- the address of the file name string descriptor (optional). If specified, the name is entered in the directory specified by the FIB.

- P3 -- the address of the word that is to receive the length of the resulting file name string (optional).

- P4 -- the address of a descriptor for a buffer that is to receive the resulting file name string (optional).

- P5 -- the address of a list of attribute descriptors (optional). If specified, the indicated attributes are read (IO$_ACCESS), or written (IO$_CREATE, IO$_DEACCESS, and IO$_MODIFY).

(See Chapter 9 for more information on these functions.)

The function-dependent arguments for IO$_READVBLK, IO$_READLBLK, IO$_WRITEVBLK, and IO$_WRITELBLK are:

- P1 -- the starting virtual address of the buffer that is to receive the data in the case of a read operation; or, in the case of a write operation, the virtual address of the buffer that is to be written on the disk.

- P2 -- the number of bytes that are to be read from the disk, or written from memory to the disk. An even number must be specified if the controller is an RK611, RL11 or RX211.

- P3 -- the starting virtual/logical disk address of the data to be transferred in the case of a read operation; or, in the case of a write operation, the disk address of the area that is to receive the data.

  In a virtual read or write, the address is expressed as a block number within the file, that is, block 1 of the file is virtual block 1. (Virtual block numbers are converted to logical block numbers via mapping windows set up by the file system ACP process.)

  In a logical read or write, the address is expressed as a block number relative to the start of the disk. For example, the first sector on the disk contains (at least the beginning of) block 0.

The function-dependent arguments for IO$_WRITECHECK, IO$_READPBLK, and IO$_WRITEPBLK are:

- P1 -- the starting virtual address of the buffer that is to receive the data in the case of a read operation; or in the case of a write operation, the starting virtual address of the buffer that is to be written on the disk.

- P2 -- the number of bytes that are to be read from the disk, or written from memory to the disk. An even number must be specified if the controller is an RK611, RL11, or RX211.

- P3 -- the starting physical disk address of the data to be read in the case of a read operation; or, in the case of a write operation, the starting physical address of the disk area that is to receive the data. The address is expressed as sector, track, and cylinder in the format shown in Figure 3-2.

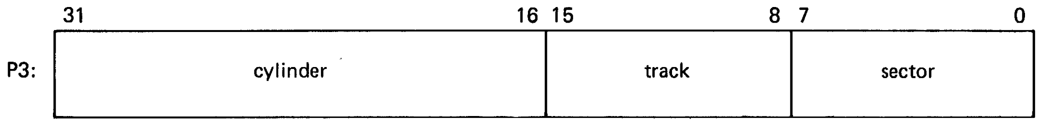


Figure 3-2 Starting Physical Address

The function-dependent argument for IO$_SEARCH is:

- P1 -- the physical disk address to position to. The address is expressed as sector, track, and cylinder in the format shown in Figure 3-2.

The function-dependent argument for IO$_SEEK is:

- P1 -- the physical cylinder number to position to. The address is expressed in the format shown in Figure 3-3.

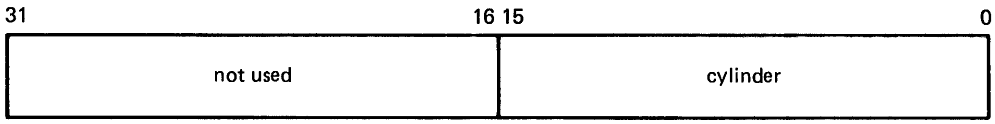


Figure 3-3 Physical Cylinder Number Format

The function-dependent argument for IO$_SETMODE and IO$_SETCHAR is:

- P1 -- the address of a quadword device characteristics descriptor

3.4.1 Read

This function reads data into a specified buffer from disk starting at a specified disk address.

VAX/VMS provides three read function codes:

- IO$_READVBLK - read virtual block
- IO$_READLBLK - read logical block
- IO$_READPBLK - read physical block

If a read virtual block function is directed to a volume that is mounted foreign, the function is converted to read logical block. If a read virtual block function is directed to a volume that is mounted structured, the volume is handled in the normal manner for a file-structured device.

Three function-dependent arguments are used with these codes: P1, P2, and P3. These arguments were described above, in the beginning of Section 3.4.

The data check function modifier (IO$M_DATACHECK) can be used with all read functions. If this modifier is specified, a data check operation is performed after the read operation has been completed. A data check operation is also performed if the volume read, or the volume on which the file resides (virtual read), has the characteristic "data check all reads." Furthermore, a data check is performed after a virtual read if the file has the attribute "data check on read." The RX01 driver does not support the data check function.

The read check function and the data check function modifier to a disk or tape can return five error codes in the I/O status block: SS$_NORMAL, SS$_CTRLERR, SS$_DRVERR, SS$_MEDOFL, and SS$_NONEXDRV. If no errors are detected, the disk or tape data is considered reliable.

The inhibit retry function modifier (IO$M_INHRETRY) can be used with all read functions. If this modifier is specified, all error recovery attempts are inhibited. IO$M_INHRETRY takes precedence over IO$M_DATACHECK. If both are specified and an error occurs, there is no attempt at error recovery and no data check operation is performed. If an error does not occur, the data check operation is performed.

3.4.2 Write

This function writes data from a specified buffer to disk starting at a specified disk address.

VAX/VMS provides three write function codes:

- IO$_WRITEVBLK - write virtual block
- IO$_WRITELBLK - write logical block
- IO$_WRITEPBLK - write physical block

If a write virtual block function is directed to a volume that is mounted foreign, the function is converted to write logical block. If a write virtual block function is directed to a volume that is mounted structured, the volume is handled in the normal manner for a file-structured device.

Three function-dependent arguments are used with these codes: P1, P2, and P3. These arguments were described above, in the beginning of Section 3.4.

The data check function modifier (IO$M_DATACHECK) can be used with all write functions. If this modifier is specified, a data check operation is performed after the write operation has been completed. A data check operation is also performed if the volume written, or the volume on which the file resides (virtual write), has the characteristic "data check all writes." Furthermore, a data check is performed after a virtual write if the file has the attribute "data check on write." The RX01 driver does not support the data check function.

The inhibit retry function modifier (IO$M_INHRETRY) can be used with all write functions. If this modifier is specified, all error recovery attempts are inhibited. IO$M_INHRETRY takes precedence over IO$M_DATACHECK. If both are specified and an error occurs, there is no attempt at error recovery and no data check operation is performed. If an error does not occur, the data check operation is performed.

3.4.3 Set Mode

Set mode operations affect the operation and characteristics of the associated disk device. VAX/VMS defines two types of set mode functions:

- Set Mode
- Set Characteristic

3.4.3.1 **Set Mode** - The Set Mode function affects the operation and characteristics of the associated disk device. Set Mode is a logical I/O function and requires the access privilege necessary to perform logical I/O. A single function code is provided:

IO$_SETMODE

This function takes the following device/function-dependent argument (other arguments are not valid):

P1 -- the address of a characteristics buffer

The P1 argument points to a quadword block shown in Figure 3-4.

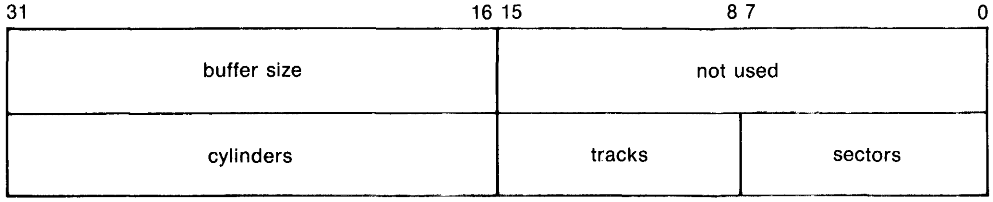


Figure 3-4 Set Mode Characteristics Buffer

The buffer size is the default for disk transfers (this default is normally 512 bytes). The second longword of the buffer contains information on the cylinder, track, and sector configuration of the particular device; that is, number of cylinders per mass storage media volume (bits 31:16), number of tracks per cylinder (bits 15:8), and number of sectors per track (bits 7:0).

3.4.3.2 **Set Characteristic** - The Set Characteristic function affects the characteristics of the associated disk device. Set Characteristic is a physical I/O function and requires the access privilege necessary to perform physical I/O functions. A single function code is provided:

IO$_SETCHAR

This function takes the following device/function-dependent argument (other arguments are not valid):

P1 -- the address of a characteristics buffer

The P1 argument points to a quadword block as shown in Figure 3-5.

31 16	15 8	7 0
buffer size	type	class
cylinders	tracks	sectors

Figure 3-5 Set Characteristic Buffer

The device class for disks is DC$_DISK. Disk types are listed in Section 3.3. The buffer size is the default for disk transfers (this default is normally 512 bytes). The second longword of the buffer contains information on the cylinder, track, and sector configuration of the particular device; that is, number of cylinders per mass storage media volume (bits 31:16), number of tracks per cylinder (bits 15:8), and number of sectors per track (bits 7:0).

3.4.4 Sense Mode

Sense mode operations obtain current disk device-dependent characteristics and return them to the caller in the second longword of the I/O status block (see Figure 3-7). VAX/VMS provides a single function code:

IO$_SENSEMODE - Sense Mode

Sense Mode is a logical I/O function and requires the access privilege necessary to perform logical I/O. No device/function-dependent arguments are used with IO$_SENSEMODE.

3.4.5 Pack Acknowledge

This function sets the volume valid bit for all disk devices. Pack acknowledge is a physical I/O function and requires the access privilege to perform physical I/O. A single function code is provided:

IO$_PACKACK

This function code takes no function-dependent arguments.

IO$_PACKACK must be the first function issued when a volume (pack, cartridge, or diskette) is placed in a disk drive. IO$_PACKACK is issued automatically when the INITIALIZE or MOUNT command language commands are issued.

3.5 I/O STATUS BLOCK

Figure 3-6 shows the I/O status block (IOSB) for all disk device QIO functions except Sense Mode. Figure 3-7 shows the I/O status block for Sense Mode. Table 3-4 lists the status returns for all functions and devices.

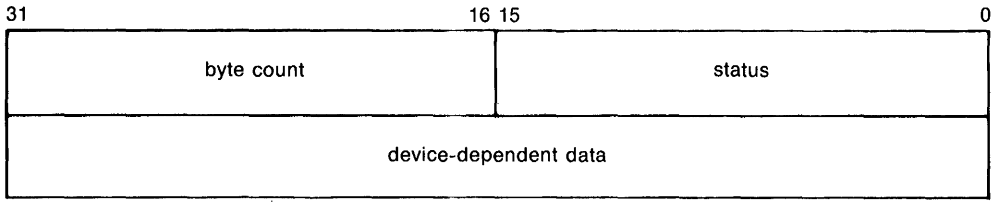


Figure 3-6 IOSB Content

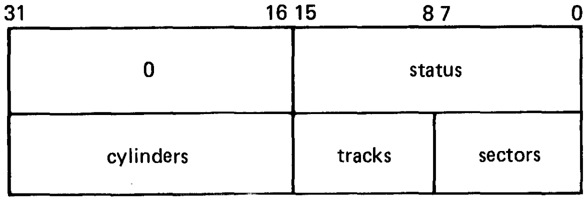


Figure 3-7 IOSB Content - Sense Mode

The byte count is the actual number of bytes transferred to or from the process buffer. Table 3-2 (in Section 3.3) lists the device-dependent data returned in the second longword.

The second longword of the I/O status block for the Sense Mode and Sense Characteristic functions returns information on the cylinder, track, and sector configuration for the particular device.

Table 3-4
Status Returns for Disk Devices

Status	Meaning
SS$_NORMAL	Successful completion of the operation specified in the QIO request. The second word of the IOSB can be examined to determine the actual number of bytes transferred to or from the buffer.
SS$_CTRLERR	Controller-related error. For example, one or more of the following conditions can cause this error: Late data Error confirmation Invalid map register Interface timeout Missed transfer Programming error Read timeout

(continued on next page)

Table 3-4 (Cont.)
Status Returns for Disk Devices

Status	Meaning
SS$_DATACHECK	Data check error. A mismatch between the data in memory and the data on disk was detected during a data check operation (see Section 3.2.1).
SS$_DRVERR	Drive-related error. For example, one or more of the following conditions can cause this error: Driver timing error Illegal function Illegal register Operation incomplete Register modify refused Write clock failure
SS$_FORMAT	Format error. Format specified by driver does not correspond to format as specified in sector headers. Disk has been formatted for another computer, such as DECsystem-20.
SS$_INBUFLEN	Invalid buffer length. The byte count must be even for UNIBUS disk devices, that is, RK07.
SS$_IVADDR	Invalid disk address error. Either an invalid starting disk address or a disk address that was incremented causes this error. This error occurs for physical read and write operations or as the result of a hardware error.
SS$_MEDOFL	Medium offline. The addressed drive currently does not have a volume mounted and on line.
SS$_NONEXDRV	Nonexistent drive. The addressed drive does not exist or the drive select plug has been removed.
SS$_PARITY	Parity error. For example, one or more of the following conditions can cause this error: Drive parity error ECC hard error Header compare error Map parity error Header CRC error MASSBUS control parity error MASSBUS data parity error

(continued on next page)

Table 3-4 (Cont.)
Status Returns for Disk Devices

Status	Meaning
SS$_UNSAFE	Drive unsafe. The addressed drive is currently unsafe and cannot perform any operation as the result of a hardware error.
SS$_VOLINV	Volume invalid. The addressed drive has not been mounted and therefore does not have volume valid set, or a status change has occurred in the drive so that it has changed to an unknown, and therefore, invalid state. All logical and virtual functions will return this status until volume valid is set. Volume valid is set when a IO$_PACKACK function is executed (usually by the MOUNT command language command) and cleared when the volume is unloaded, the respective drive changes to an unknown state, or the power fails. The driver automatically sets volume valid when the proper volume is mounted and/or power is restored.
SS$_WASECC	Data check not performed. The function was a read data that was completed successfully by applying one or more ECC corrections. The specified data check, however, was not performed.
SS$_WRITLCK	Write lock error. An attempt was made to write on a write locked drive. Volume is hardware protected.

3.6 PROGRAMMING EXAMPLE

The following program provides an example of optimizing access time to a disk file. The program creates a file using VAX-11 RMS, stores information concerning the file, and closes the file. The program then accesses the file and reads and writes to the file using the Queue I/O system service.

```
        .TITLE  Disk Driver Programming Example
        .IDENT  /01/


;
; Define necessary symbols
;

        $FIBDEF                         ;Define File Information Block Offsets
        $IODEF                          ;Define I/O function codes
        $RMSDEF                         ;Define RMS-32 Return Status Values

;
; Local storage
;
; Define number of records to be processed
;

NUM_RECS=100                            ;One hundred records

;
; Allocate storage for necessary data structures
;
; Allocate File Access Block
;
;       A file access block is required by RMS-32 to open and close a file.
;

FAB_BLOCK:                              ;
        $FAB    ALQ = 100,-             ;Initial file size is to be 100 blocks
                FAC = PUT,-             ;File Access Type is output
                FNA = FILE_NAME,-       ;File name string address
                FNS = FILE_SIZE,-       ;File name string size
                FOP = CTG,-             ;File is to be contiguous
                MRS = 512,-             ;Maximum record size is 512 bytes
                NAM = NAM_BLOCK,-       ;File name block address
                ORG = SEQ,-             ;File organization is to be sequential
                RFM = FIX               ;Record format is fixed length

;
; Allocate file information block
;
;       A file information block is required as an argument in the Queue I/O
;       system service call that accesses a file.
;

FIB_BLOCK:                              ;
        .BLKB   FIB$K_LENGTH            ;

;
; Allocate file information block descriptor
;

FIB_DESCR:                              ;
        .LONG   FIB$K_LENGTH            ;Length of file information block
        .LONG   FIB_BLOCK               ;Address of file information block

;
; Allocate File Name Block
;
;       A file name block is required by RMS-32 to return information concerning
;       a file (e.g. the resultant filename string after logical name translation
;       and defaults have been applied).
;

NAM_BLOCK:                              ;
        $NAM                            ;

;
; Allocate Record Access Block
;
;       A record access block is required by RMS-32 for record operations on a
;       file.
;

RAB_BLOCK:                              ;
        $RAB    FAB = FAB_BLOCK,-       ;File access block address
                RAC = SEQ,-             ;Record access is to be sequential
                RBF = RECORD_BUFFER,-   ;Record buffer address
                RSZ = 512               ;Record buffer size

;
; Allocate direct access buffer
;

BLOCK_BUFFER:                           ;
        .BLKB   1024                    ;Direct access buffer is 1024 bytes

;
; Allocate space to store channel number returned by the Assign Channel system
; service
```

```
;
DEVICE_CHANNEL:                         ;
        .BLKW   1                       ;

;
; Allocate device name string and descriptor
;
DEVICE_DESCR:                           ;
        .LONG   20$-10$                 ;Length of device name string
        .LONG   10$                     ;Address of device name string
10$:    .ASCII  /SYS$DISK/              ;Device on which created file will reside
20$:                                    ;Reference label to calculate length

;
; Allocate file name string and define string length symbol
;

FILE_NAME:                              ;
        .ASCII  /SYS$DISK:MYDATAFIL.DAT/ ;File name string
FILE_SIZE=.-FILE_NAME                   ;File name string length

;
; Allocate I/O status quadword storage
;
IO_STATUS:                              ;
        .BLKQ   1                       ;

;
; Allocate output record buffer
;
RECORD_BUFFER:                          ;
        .BLKB   512                     ;Record buffer is 512 bytes

;
; Program starting point
;
; The general logic of the program is to create a file called MYDATAFIL.DAT
; using RMS-32, store information concerning the file, write 100 records each
; of which contains its record number in every byte, close the file, and then
; access and read and write the file directly using the Queue I/O system service.
; If any errors are detected, the program returns to its caller with the final
; error status in register R0.
;

        .ENTRY  DISK_EXAMPLE,^M<R2,R3,R4,R5,R6> ;Program starting address

;
; First create the file and open it using RMS-32
;

PART_1:                                 ;First part of example
        $CREATE FAB = FAB_BLOCK         ;Create and open file
        BLBC    R0,20$                  ;If low bit clear, creation failure

;
; Second connect the record access block to the created file
;

        $CONNECT RAB = RAB_BLOCK        ;Connect the record access block
        BLBC    R0,30$                  ;If low bit clear, connection failure

;
; Now write 100 records each containing its record number
;

        MOVZBL  #NUM_RECS,R6            ;Set record write loop count

;
; Fill each byte of the record to be written with its record number
;


10$:    SUBB3   R6,#NUM_RECS+1,R5       ;Calculate record number

        MOVC5   #0,(R6),R5,#512,RECORD_BUFFER ;Fill record buffer


;
; Next write the record into the newly created file using RMS-32
;

        $PUT    RAB = RAB_BLOCK         ;Put record in file
        BLBC    R0,30$                  ;If low bit clear, put failure
        SOBGTR  R6,10$                  ;Any more records to write?

;
; The file creation part of the example is almost complete. All that remains to
; be done is to store the file information returned by RMS-32 and close the file.
;

        MOVW    NAM_BLOCK+NAM$W_FID,FIB_BLOCK+FIB$W_FID ;Save file identification
        MOVW    NAM_BLOCK+NAM$W_FID+2,FIB_BLOCK+FIB$W_FID+2 ;Save sequence number
        MOVW    NAM_BLOCK+NAM$W_FID+4,FIB_BLOCK+FIB$W_FID+4 ;Save relative volume
        $CLOSE  FAB = FAB_BLOCK         ;Close file
        BLBS    R0,PART_2               ;If low bit set, successful close
20$:    RET                             ;Return with RMS error status
```

```
;
; Record stream connection or put record failure
;
; Close file and return status
;

30$:    PUSHL   R0                          ;Save error status
        $CLOSE  FAB = FAB_BLOCK             ;Close file
        POPL    R0                          ;Retrieve error status
        RET                                 ;Return with RMS error status

;
; The second part of the example illustrates accessing the previously created
; file directly using the Queue I/O system service, randomly reading and writing
; various parts of the file, and then deaccessing the file.
;
; First assign a channel to the appropriate device and access the file
;

PART_2:                                     ;
        $ASSIGN_S DEVNAM = DEVICE_DESCR,-   ;Assign a channel to file device
                CHAN = DEVICE_CHANNEL       ;
        BLBC    R0,20$                      ;If low bit clear, assignment failure
        MOVL    #FIB$M_NOWRITE!FIB$M_WRITE,- ;Set for read/write access
                FIB_BLOCK+FIB$L_ACCTL       ;
        $QIOW_S CHAN = DEVICE_CHANNEL,-     ;Access file on device channel
                FUNC = #IO$_ACCESS!IO$M_ACCESS,- ;I/O function is access file
                IOSB = IO_STATUS,-          ;Address of I/O status quadword
                P1 = FIB_DESCR              ;Address of information block descriptor
        BLBC    R0,10$                      ;If low bit clear, access failure
        MOVZWL  IO_STATUS,R0                ;Get final I/O completion status


        BLBS    R0,30$                      ;If low bit set, successful I/O function
10$:    PUSHL   R0                          ;Save error status
        $DASSGN_S CHAN = DEVICE_CHANNEL     ;Deassign file device channel
        POPL    R0                          ;Retrieve error status
20$:    RET                                 ;Return with I/O error status

;
; The file is now ready to be read and written randomly. Since the records are
; fixed length and exactly one block long, the record number corresponds to the
; virtual block number of the record in the file. Thus a particular record can
; be read or written simply by specifying its record number in the file.
;
; The following code reads 2 records at a time and checks to see that they contain
; their respective record numbers in every byte. The records are then written back
; into the file in reverse order. This results in record 1 having the old contents
; of record 2 and record 2 the old contents of record 1 ,etc. After the example
; has been run, it is suggested that the file dump utility be used to verify this
; fact.
;

30$:    MOVZBL  #1,R6                       ;Set starting record (block) number

;
; Read next 2 records into block buffer
;

40$:    $QIOW_S CHAN = DEVICE_CHANNEL,-     ;Read next 2 records from file channel
                FUNC = #IO$_READVBLK,-      ;I/O function is read virtual block
                IOSB = IO_STATUS,-          ;Address of I/O status quadword
                P1 = BLOCK_BUFFER,-         ;Address of I/O buffer
                P2 = #1024,-                ;Size of I/O buffer
                P3 = R6                     ;Starting virtual block of transfer
        BSBB    50$                         ;Check I/O completion status

;
; Check each record to make sure it contains the correct data
;

        SKPC    R6,#512,BLOCK_BUFFER        ;Skip over equal record numbers in data

        BNEQ    60$                         ;If not equal, data match failure
        ADDL3   #1,R6,R5                    ;Calculate even record number

        SKPC    R5,#512,BLOCK_BUFFER+512 ;Skip over equal record numbers in data

        BNEQ    60$                         ;If not equal, data match failure

;
; Record data matches
;
; write records in reverse order in file
;

        $QIOW_S CHAN = DEVICE_CHANNEL,-     ;write even numbered record in odd slot
                FUNC = #IO$_WRITEVBLK,-     ;I/O function is write virtual block
                IOSB = IO_STATUS,-          ;Address of I/O status quadword
                P1 = BLOCK_BUFFER+512,-     ;Address of even record buffer
                P2 = #512,-             ;Length of even record buffer
                P3 = R6                 ;Record number of odd record
        BSBB    50$                     ;Check I/O completion status
        ADDL3   #1,R6,R5                ;Calculate even record number

        $QIOW_S CHAN = DEVICE_CHANNEL,- ;Write odd numbered record in even slot
                FUNC = #IO$_WRITEVBLK,- ;I/O function is write virtual block
                IOSB = IO_STATUS,-      ;Address of I/O status quadword
                P1 = BLOCK_BUFFER,-     ;Address of odd record buffer
                P2 = #512,-             ;Length of odd record buffer
                P3 = R5                 ;Record number of even record
        BSBB    50$                     ;Check I/O completion status
        ACBB    #NUM_RECS-1,#2,R6,40$   ;Any more records to be read?

        BRB     70$                     ;

;
; Check I/O completion status
;

50$:    BLBC    R0,70$                  ;If low bit clear, service failure
        MOVZWL  IO_STATUS,R0            ;Get final I/O completion status
        BLBC    R0,70$                  ;If low bit clear, I/O function failure
        RSB                             ;

;
; Record number mismatch in data
;

60$:    MNEGL   #4,R0                   ;Set dummy error status value


;
; All records have been read, verified, and odd/even pairs inverted
;

70$:    PUSHL   R0                      ;Save final status
        $QIOW_S CHAN = DEVICE_CHANNEL,- ;Deaccess file
                FUNC = #IO$_DEACCESS    ;I/O function is deaccess file
        $DASSGN_S CHAN = DEVICE_CHANNEL ;Deassign file device channel
        POPL    R0                      ;Retrieve final status
        RET                             ;

        .END    DISK_EXAMPLE
```

CHAPTER 4

MAGNETIC TAPE DRIVER

This chapter describes the use of the VAX/VMS magnetic tape driver. This driver supports the devices listed in Table 4-1 and detailed in Section 4.1.

Table 4-1
Magnetic Tape Devices

Model	No. of Tracks	Recording Density (bpi)	Tape Speed (ips)	Max. Data Transfer Rate in Bytes Per Second	Recording Method
TE16	9	800 or 1600	45	36,000 (for 800 bpi); 72,000 (for 1600 bpi)	NRZI or PE[1]
TS11	9	800 or 1600	45	36,000 (for 800 bpi); 72,000 (for 1600 bpi)	NRZI or PE
TU45	9	800 or 1600	75	60,000 (for 800 bpi) 120,000 (for 1600 bpi)	NRZI or PE
TU77	9	800 or 1600	125	100,000 (for 800 bpi) 200,000 (for 1600 bpi)	NRZI or PE

[1] NRZI = non-return-to-zero-inverted; PE = phase encoded.

4.1 SUPPORTED MAGNETIC TAPE DEVICES

The following sections describe the magnetic tape drives in greater detail.

4.1.1 TE16 Magnetic Tape Drive

The TE16 magnetic tape drive holds one, 2400-foot, 9-track reel with a capacity of 40 million characters. The drive reads data at 45 inches per second with an average transfer time of 14 microseconds per byte at the 1600 bpi density. Up to eight drives can be connected to each TM03 controller.

4.1.2 TS11 Magnetic Tape Subsystem

The TS11 Magnetic Tape is a phase-encoded, 9-track magnetic tape subsystem that operates under microprocessor control. The TS11 consists of one TS11 controller and one TS04 drive.

4.1.3 TU45 and TU77 Magnetic Tape System

The TU45 and TU77 are phase-encoded, 9-track magnetic tape systems with a capacity of 40 million characters. Tape density and character format are program selectable.

4.2 DRIVER FEATURES AND CAPABILITIES

The VAX/VMS magnetic tape driver provides the following features:

- Multiple master adapters and slave formatters
- Different types of devices on a single MASSBUS adapter; for example, RP05 disk and TM03 tape formatter
- Reverse read and reverse data check functions (not for TS11)
- Data checks on a per-request, per-file, and/or per-volume basis (not for TS11)
- Full recovery from power failure for online drives with volumes mounted, including repositioning by the driver
- Extensive error recovery algorithms; for example, non-return-to-zero-inverted (NRZI) error correction
- Logging of device errors in a file that may be displayed by field service or customer personnel
- Online diagnostic support for drive level diagnostics

The following sections describe master and slave controllers, and data check and error recovery capabilities in greater detail.

4.2.1 Master Adapters and Slave Formatters

VAX/VMS supports the use of multiple master adapters of the same type on a system. For example, more than one MASSBUS adapter (MBA) can be used on the same system. A master adapter is a device controller capable of performing and synchronizing data transfers between memory and one or more slave formatters.

VAX/VMS also supports the use of multiple slave formatters per master adapter on a system. For example, more than one TM03 Magnetic Tape Formatter per MBA can be used on a system. A slave formatter accepts data and/or commands from a master adapter and directs the operation of one or more slave drives. The TM03 is a slave formatter. The TE16 Magnetic Tape Transport is a slave drive.

4.2.2 Data Check

A data check is made after successful completion of an I/O operation to compare the data in memory with that on the tape. After a write or read (forward) operation, the tape drive backspaces and then performs a write check data operation. After a read in the reverse direction, the tape drive forward spaces and then performs a write check data reverse operation. With the exception of the TS11, magnetic tape drivers support data checks at three levels:

- Per request -- Users can specify the data check function modifier (IO$M_DATACHECK) on a read logical block, write logical block, read virtual block, write virtual block, read physical block, or write physical block I/O function.
- Per volume -- Users can specify the characteristics "data check all reads" and/or "data check all writes" when the volume is mounted. The VAX/VMS Command Language User's Guide describes volume mounting and dismounting.
- Per file -- Users can specify the file attributes "data check on read" or "data check on write." File access attributes are specified when the file is accessed. Chapter 9 of this manual and the VAX-11 Record Management Services Reference Manual describe file access.

4.2.3 Error Recovery

Error recovery in VAX/VMS is aimed at performing all possible operations to complete an I/O operation successfully. Magnetic tape error recovery operations fall into two categories:

- Handling special conditions such as power failure and interrupt timeout
- Retrying nonfatal controller and/or drive errors

The error recovery algorithm uses a combination of these two types of error recovery operations.

Power failure recovery consists of waiting for mounted drives to be unloaded by the operator. When the drives are reloaded, the driver automatically spaces to the position held before the power failure. The I/O operation that was in progress at the time of the power failure is then re-executed. To solicit reloading of mounted drives, device not ready messages are sent to the operator console after a power failure.

Device timeout is treated as a fatal error with a loss of tape position. A tape on which a timeout has occurred must be dismounted and rewound before the drive position can be established.

Nonfatal controller/drive errors are simply re-executed up to 16 times before returning a fatal error. The tape is repositioned as necessary before each retry.

All normal error recovery (nonspecial conditions) can be inhibited by specifying the inhibit retry function modifier (IO$M_INHRETRY). If any error occurs and this modifier is specified, the operation is immediately terminated, and a failure status is returned. This modifier has no effect on power failure and timeout recovery.

Up to 16 extended interrecord gaps can be written during the error recovery for a write operation. Except for the TS11, writing of these gaps can be suppressed by specifying the inhibit extended interrecord gap function modifier (IO$M_INHEXTGAP).

4.3 DEVICE INFORMATION

Users can obtain information on device characteristics by using the $GETCHN and $GETDEV system services (see Section 1.10). The information is returned in a user-specified buffer shown in Figure 4-1. Only the first three longwords of the buffer are shown in Figure 4-1 (Figure 1-8 shows the entire buffer).

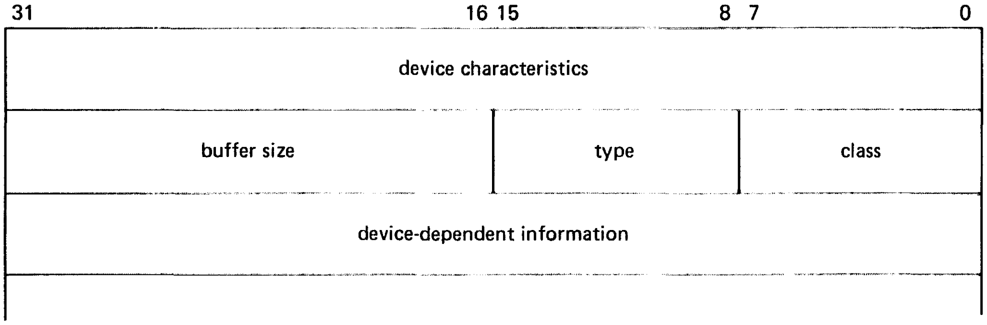


Figure 4-1 Magnetic Tape Information

The device characteristics returned in the first longword are listed in Table 4-2.

Table 4-2
Magnetic Tape Device-Independent Characteristics

Dynamic Bits[1] (Conditionally Set)	Meaning
DEV$M_AVL	Device is on line and available
DEV$M_FOR	Foreign volume
DEV$M_MNT	Volume mounted
DEV$M_RCK	Perform data check all reads
DEV$M_WCK	Perform data check all writes
Static Bits[1] (Always Set)	
DEV$M_FOD	File-oriented device
DEV$M_IDV	Device is capable of input
DEV$M_ODV	Device is capable of output
DEV$M_SQD	Device is sequential access

[1] Defined by the $DEVDEF macro.

The second longword contains information on device class and type, and the buffer size. The device class for tapes in DC$_TAPE. The device type is DT$_TE16 for the TE16 and DT$_TS11 for the TS11.

The $DCDEF macro defines the device type and class names. The buffer size is the default to be used for tape transfers (this default is normally 2048 bytes).

The third longword contains device-dependent information. Table 4-3 lists this information. The $MTDEF macro defines the values listed.

Table 4-3
Device-Dependent Information for Tape Devices

Value	Meaning	
MT$M_LOST	If set, the current tape position is unknown.	
MT$M_HWL	If set, the selected drive is hardware write-locked.	
MT$M_EOT	If set, an end-of-tape (EOT) condition was encountered by the last operation to move tape in the forward direction.	
MT$M_EOF	If set, a tape mark was encountered by the last operation to move tape.	
MT$M_BOT	If set, a beginning-of-tape (BOT) marker was encountered by the last operation to move tape in the reverse direction.	
MT$M_PARITY	If set, all data transfers are performed with even parity. If clear (normal case), all data transfers are performed with odd parity. Only NRZI recording at 800 bpi can have even parity.	
MT$V_DENSITY MT$S_DENSITY	Specifies the density at which all data transfer operations are performed. Possible density values are:	
	MT$K_PE_1600	Phase encoded, 1600 bpi.
	MT$K_NRZI_800	Non-return-to-zero-inverted, 800 bpi.
MT$V_FORMAT MT$S_FORMAT	Specifies the format in which all data transfers are performed. A possible format value is:	
	MT$K_NORMAL11	Normal PDP-11 format. Data bytes are recorded sequentially on tape with each byte occupying exactly one frame.

4.4 MAGNETIC TAPE FUNCTION CODES

The VAX/VMS magnetic tape driver can perform logical, virtual, and physical I/O functions.

Logical and physical I/O functions to magnetic tape devices allow sequential access to volume storage and require only that the requesting process have direct access to the device. Virtual I/O functions require intervention by an ancillary control process (ACP) and must be executed in a prescribed order. The normal procedure is to create a file and access it. Information is then written to the file and the file is deaccessed. The file is subsequently accessed, the information is read, and the file is deaccessed. The file can be written over when the information it contains is no longer useful and the file has expired.

Any number of bytes (up to a maximum of 64K) can be read from or written into a single block by a single request. The number of bytes itself has no effect on the applicable quotas (direct I/O, buffered I/O, and AST). Reading or writing any number of bytes subtracts the same amount from a quota.

The volume to which a logical or virtual function is directed must be mounted in order for the function to actually be executed. If it is not, either a device not mounted or invalid volume status is returned in the I/O status block.

Table 4-4 lists the logical, virtual, and physical magnetic tape I/O functions and their function codes. These functions are described in more detail in the following paragraphs. Chapter 9 describes the QIO level interface to the magnetic tape device ACP.

Table 4-4
Magnetic Tape I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_CREATE P1,[P2],- [P3],[P4],[P5]	V	IO$M_CREATE IO$M_ACCESS	Create a file
IO$_ACCESS P1,[P2],- [P3],[P4],[P5]	V	IO$M_CREATE IO$M_ACCESS	Search a tape for a specified file and access the file if found and IO$M_ACCESS is set. If the file is not found and IO$M_CREATE is set, create a file at end-of-tape
IO$_DEACCESS P1,[P2],- [P3],[P4],[P5]	V		Deaccess a file and, if the file has been written, write out trailer records
IO$_MODIFY P1,[P2],- [P3],[P4],[P5]	V		Write user labels

[1] V = virtual; L = logical; P = physical.

(continued on next page)

Table 4-4 (Cont.)
Magnetic Tape I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_READVBLK P1,P2	V	IO$M_DATACHECK[2] IO$M_INHRETRY IO$M_REVERSE	Read virtual block
IO$_READLBLK P1,P2	L	IO$M_DATACHECK[2] IO$M_INHRETRY IO$M_REVERSE	Read logical block
IO$_READPBLK P1,P2	P	IO$M_DATACHECK[2] IO$M_INHRETRY IO$M_REVERSE	Read physical block
IO$_WRITEVBLK P1,P2	V	IO$M_DATACHECK[2] IO$M_INHRETRY IO$M_INHEXTGAP	Write virtual block
IO$_WRITELBLK P1,P2	L	IO$M_DATACHECK[2] IO$M_INHRETRY IO$M_INHEXTGAP	Write logical block
IO$_WRITEPBLK P1,P2	P	IO$M_DATACHECK[2] IO$M_INHRETRY IO$M_INHEXTGAP	Write physical block
IO$_REWIND	L	IO$M_INHRETRY IO$M_NOWAIT	Reposition tape to the beginning of tape (BOT) marker
IO$_SKIPFILE P1	L	IO$M_INHRETRY	Skip past a specified number of tape marks in either a forward or reverse direction
IO$_SKIPRECORD P1	L	IO$M_INHRETRY	Skip past a specified number of blocks in either a forward or reverse direction
IO$_WRITEOF	L	IO$M_INHRETRY IO$M_INHEXTGAP	Write an extended interrecord gap followed by a tape mark
IO$_REWINDOFF	L	IO$M_INHRETRY IO$M_NOWAIT	Rewind and unload the tape on the selected drive

1. V = virtual; L = logical; P = physical.

2. Not for TS11

(continued on next page)

Table 4-4 (Cont.)
Magnetic Tape I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_SENSEMODE	L	IO$M_INHRETRY	Sense the tape characteristics and return them in the I/O status block
IO$_SETMODE P1	L		Set tape characteristics for subsequent operations
IO$_SETCHAR P1	P		Set tape characteristics for subsequent operations
IO$_ACPCONTROL P1,[P2],-[P3],[P4],[P5]	V	IO$M_DMOUNT	Perform miscellaneous control functions (see Section 9.3)
IO$_MOUNT	V		Informs ACP when volume is mounted; requires mount privilege.

[1] V = virtual; L = logical; P = physical.

The function-dependent arguments for IO$_CREATE, IO$_ACCESS, IO$_DEACCESS, and IO$_MODIFY are:

- P1 -- the address of the File Information Block (FIB) descriptor.
- P2 -- the address of the file name string descriptor (optional). If specified with IO$_ACCESS, the name identifies the file being sought. If specified with IO$_CREATE, the name is the name of the created file.
- P3 -- the address of the word that is to receive the length of the resultant file name string (optional).
- P4 -- the address of a descriptor for a buffer that is to receive the resultant file name string (optional).
- P5 -- the address of a list of attribute descriptors (optional). If specified with IO$_ACCESS, the attributes of the file are returned to the user. If specified with IO$_CREATE, P5 is the address of the attribute descriptor list for the new file. All file attributes for IO$_MODIFY are ignored.

(See Chapter 9 for more information on these functions.)

The function-dependent arguments for IO$_READVBLK, IO$_READLBLK, IO$_READPBLK, IO$_WRITEVBLK, IO$_WRITELBLK, and IO$_WRITEPBLK are:

- P1 -- the starting virtual address of the buffer that is to receive the data in the case of a read operation; or, in the case of a write operation, the virtual address of the buffer that is to be written on the tape.
- P2 -- the number of bytes that are to be read from the tape, or written from memory to the tape.

The function-dependent argument for IO$_SKIPFILE and IO$_SKIPRECORD is:

- P1 -- the number of tape marks to skip over in the case of a skip file operation; or, in the case of a skip record operation, the number of blocks to skip over. If a positive number is specified, the tape moves forward; if a negative number is specified, the tape moves in reverse. (The maximum number of tape marks or records that P1 can specify is 32,767.)

4.4.1 Read

This function reads data into a specified buffer in the forward or reverse direction starting at the next block position.

VAX/VMS provides three read function codes:

- IO$_READVBLK - read virtual block
- IO$_READLBLK - read logical block
- IO$_READPBLK - read physical block

A read virtual block function to a volume that is mounted foreign is converted to a read logical block function. A read virtual block function to a volume that is mounted structured is handled in the normal manner for a file-structured device.

If the reverse function modifier (IO$M_REVERSE) is specified, the read operation is performed in the reverse direction instead of the forward direction.

The data check function modifier (IO$M_DATACHECK) can be used with all read functions. If this modifier is specified, a data check operation is performed after the read data operation has been completed. (A space reverse or space forward is performed between the read and the data check operation.) A data check operation is also performed if the volume read, or the volume on which the file resides (virtual read), has the characteristic "data check all reads." Furthermore, a data check is performed after a virtual read if the file has the attribute "data check on read."

If a read physical block or read logical block operation is performed and the reverse function modifies IO$M_REVERSE is not specified, an end-of-tape status is returned if either of the following conditions occur and no other error condition exists:

- The tape is positioned past the end-of-tape position at the start of the read operation.
- The tape enters the end-of-tape region as a result of the read operation.

The transferred byte count reflects the actual number of bytes read. If a read in the reverse direction is performed when the tape is positioned past the end-of-tape position, an end-of-tape status is not returned.

If a tape mark is read during a logical or physical read operation in either the forward or reverse direction, an end-of-file status is returned if any of the following conditions exist:

- The tape is positioned past the end-of-tape position at the start of the read operation.
- The tape enters the end-of-tape region as a result of the read operation.
- A tape mark is read as a result of a read operation but the tape does not enter the end-of-tape region.

An end-of-file status is also returned if a read operation in the reverse direction is attempted when the tape is positioned at the BOT marker. All conditions that cause an end-of-file status result in a transferred byte count of zero.

If an attempt is made during a logical or physical read operation to read a block that is larger than the specified memory buffer, a data overrun status is returned. Only the first part of the block is read into the specified buffer. (Only the latter part of the block is read into the buffer on a read in the reverse direction.) The transferred byte count is equal to the actual size of the block. Read reverse starts at the top of the buffer. Thus, the start of the block is at P1 plus P2 minus the length read.

It is not possible to read a block that is less than 14 bytes in length. Such records are termed "noise blocks" and are completely ignored by the driver.

4.4.2 Write

This function writes data from a specified buffer to tape in the forward direction starting at the next block position.

VAX/VMS provides three write function codes:

- IO$_WRITEVBLK - write virtual block
- IO$_WRITELBLK - write logical block
- IO$_WRITEPBLK - write physical block

If a write virtual block function is to a volume that is mounted foreign, it is converted to a write logical block function. If a write virtual block function is to a volume that is mounted structured, it is handled in the normal manner for a file-structured device.

The data check function modifier (IO$M_DATACHECK) can be used with all write functions. If this modifier is specified, a data check operation is performed after the write data operation has been completed. (A space reverse is performed between the write and the data check operation.) A data check operation is also performed if the volume written, or the volume on which the file resides (virtual write), has the characteristic "data check all writes." Furthermore, a data check is performed after a virtual write if the file has the attribute "data check on write."

A data check operation is also forced by the driver when an error occurs during a write operation. This ensures that the data can be reread.

If a write physical block or write logical block operation is performed, an end-of-tape status is returned if either of the following conditions occurs and no other error condition exists:

- The tape is positioned past the end-of-tape position at the start of the write operation.
- The tape enters the end-of-tape region as a result of the write operation.

(The transferred byte count reflects the size of the block written.)

It is not possible to write a block less than 14 bytes in length. An attempt to do so results in the return of a bad parameter status for the QIO request.

4.4.3 **Rewind**

This function repositions the tape to the beginning-of-tape (BOT) marker. If the IO$M_NOWAIT function modifier is specified, the I/O operation is completed when the rewind is initiated. Otherwise, I/O completion does not occur until the tape is positioned at the BOT marker. IO$_REWIND has no function-dependent arguments.

4.4.4 **Skip File**

This logical I/O function skips past a specified number of tape marks in either a forward or reverse direction. A function-dependent argument (P1) is provided to specify the number of tape marks to be skipped, as shown in Figure 4-2. If a positive file count is specified, the tape moves forward; if a negative file count is specified, the tape moves in reverse. (The actual number of files skipped is returned in the I/O status block.)

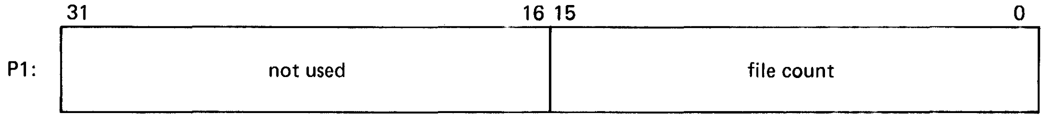


Figure 4-2 IO$_SKIPFILE Argument

Only tape marks (when the tape moves in either direction) and the BOT marker (when the tape moves in reverse) are counted during a skip file operation. The BOT marker terminates a skip file function in the reverse direction. The end-of-tape (EOT) marker does not terminate a skip file function in either the forward or reverse direction. Note that a negative skip file function leaves the tape positioned just before a tape mark, that is, at the end of a file, unless the BOT marker is encountered, whereas a positive skip file function leaves the tape positioned just past the tape mark.

4.4.5 Skip Record

The skip record function skips past a specified number of physical tape blocks in either a forward or reverse direction. A device/function-dependent argument (Pl) specifies the number of blocks to skip, as shown in Figure 4-3. If a positive block count is specified, the tape moves forward; if a negative block count is specified, the tape moves in reverse. (The actual number of blocks skipped is returned in the I/O status block.)

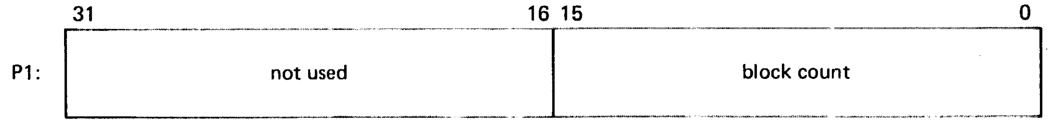


Figure 4-3 IO$_SKIPRECORD Argument

Skip record is terminated by end-of-file when the tape moves in either direction, by the BOT marker when the tape moves in reverse, and by the EOT marker when the tape moves forward.

4.4.6 Write End-of-File

This function writes an extended interrecord gap (of approximately 3 inches for NRZI recording and 1.5 inches for PE recording) followed by a tape mark. No device/function-dependent arguments are used with IO$_WRITEOF.

An end-of-tape status is returned in the I/O status block if either of the following conditions is present and no other error conditions occur:

- A write end-of-file function is executed while the tape is positioned past the EOT marker.

- A write end-of-file function causes the tape position to enter the end-of-tape region.

4.4.7 Rewind Offline

The rewind offline function rewinds and unloads the tape on the selected drive. If the IO$M_NOWAIT function modifier is specified, the I/O operation is completed as soon as the rewind is initiated. No device/function-dependent arguments are used with IO$_REWINDOFF.

4.4.8 Sense Tape Mode

This function senses the current device-dependent tape characteristics and returns them to the caller in the second longword of the I/O status block (see Table 4-3). The contents of the second longword are identical to the device-dependent information shown in Figure 4-1. No device/function-dependent arguments are used with IO$_SENSEMODE.

4.4.9 Set Mode

Set mode operations affect the operation and characteristics of the associated magnetic tape device. VAX/VMS defines two types of set mode functions:

- Set Mode
- Set Characteristic

4.4.9.1 **Set Mode** - The Set Mode function affects the characteristics of the associated tape device. Set Mode is a logical I/O function and requires the access privilege necessary to perform logical I/O. A single function code is provided:

IO$_SETMODE

This function takes the following device/function-dependent argument (other arguments are ignored):

P1 -- the address of a quadword characteristics buffer

Figure 4-4 shows the quadword Set Mode characteristics buffer.

31 16	15 0
buffer size	not used
tape characteristics	

Figure 4-4 Set Mode Characteristics Buffer

Table 4-5 lists the tape characteristics and their meanings. The $MTDEF macro defines the symbols listed.

Table 4-5
Set Mode and Set Characteristic Magnetic Tape Characteristics

MT$M_PARITY	If set, all data transfers are performed with even parity. If clear (normal case), all data transfers are performed with odd parity. Even parity can be selected only for NRZI recording at 800 bpi. Even parity cannot be selected for phase encoded recording (tape density is MT$K_PE_1600) and is ignored.

(continued on next page)

Table 4-5 (Cont.)
Set Mode and Set Characteristic Magnetic Tape Characteristics

MT$V_DENSITY MT$S_DENSITY	Specifies the density at which all data transfers are performed. Tape density can be set only when the selected drive's tape position is at the BOT marker. Possible density values are:	
	MT$K_DEFAULT	Default system density
	MT$K_PE_1600	Phase encoded, 1600 bpi
	MT$K_NRZI_800	Non-return-to-zero-inverted, 800 bpi
MT$V_FORMAT MT$S_FORMAT	Specifies the format in which all data transfers are performed. Possible format values are:	
	MT$K_DEFAULT	Default system format
	MT$K_NORMAL11	Normal PDP-11 format. Data bytes are recorded sequentially on tape with each byte occupying exactly one frame

4.4.9.2 **Set Characteristic** - The Set Characteristic function also affects the characteristics of the associated tape device. Set Characteristic is a physical I/O function and requires the access privilege necessary to perform physical I/O functions. A single function code is provided:

IO$_SETCHAR

This function takes the following device/function-dependent argument (other arguments are not valid):

P1 -- the address of a quadword characteristics buffer

Figure 4-5 shows the quadword Set Characteristic characteristics buffer.

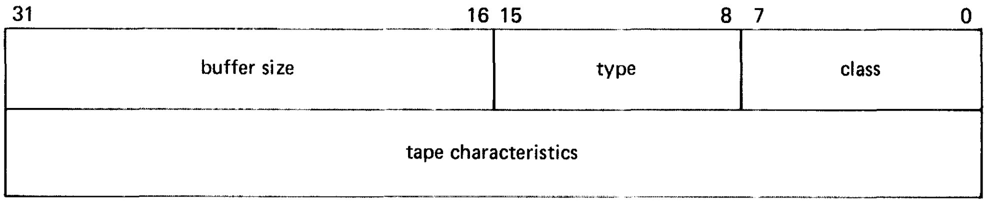


Figure 4-5 Set Characteristic Buffer

The first longword contains information on device class and type, and the buffer size. The device class for tapes is DC$_TAPE. The device type is DT$_TE16.

The $DCDEF macro defines the device type and class names. The buffer size is the default to be used for tape transfers (this default is normally 2048 bytes).

Table 4-5 lists the tape characteristics for the Set Characteristic function.

4.5 I/O STATUS BLOCK

The I/O status block (IOSB) for QIO functions on magnetic tape devices is shown in Figure 4-6. Table 4-6 lists the status returns for these functions. Table 4-3 (in Section 4.3) lists the device-dependent data returned in the second longword. The IO$_SENSEMODE function can be used to return this data.

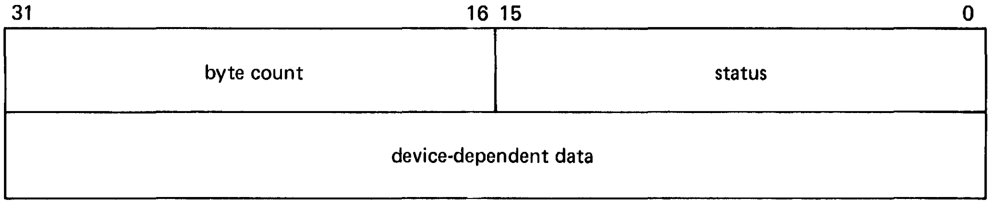


Figure 4-6 IOSB Content

The byte count is the actual number of bytes transferred to or from the process buffer or the number of files or blocks skipped.

Table 4-6
Status Returns for Tape Devices

Status	Meaning
SS$_NORMAL	Successful completion of the operation specified in the QIO request. The second word of the IOSB can be examined to determine the actual number of bytes transferred to or from the buffer or the number of files or blocks skipped.
SS$_CTRLERR	Controller-related error. One or more of the following conditions can cause this error: Data late Error confirmation Invalid map register Interface timeout Missed transfer Programming error Read timeout
SS$_DATACHECK	Write check error. A mismatch between the data in memory and the data on tape was detected during a write check operation. (See Section 4.2.1)

(continued on next page)

Table 4-6 (Cont.)
Status Returns for Tape Devices

Status	Meaning
SS$_DRVERR	Drive-related error. One or more of the following conditions can cause this error: Drive timing error Illegal function Illegal register Operation incomplete Register modify refused Nonexecutable function Unrecovered retriable error
SS$_ENDOFFILE	End-of-file condition. A tape mark was encountered during the operation. For data transfer functions, the byte count is 0; for skip record functions, the count is the number of blocks skipped.
SS$_ENDOFTAPE	End-of-tape condition. This is a normal completion and is typically treated as such. The end of an input tape is denoted by an end-of-tape marker. If this marker is encountered during an operation in the forward direction, it may be necessary to modify the source program to respond to the condition.
SS$_ENDOFVOLUME	End of volume. Two consecutive tape marks were detected during a skip file operation. This return is also used as a logical end-of-tape indicator. If an ASCII standard tape is mounted foreign, this return may only indicate an empty file within the volume and not the end of volume.
SS$_FORMAT	Format error. Format specified by last set tape characteristics function is not implemented in slave controller.
SS$_MEDOFL	Medium offline. The addressed drive currently does not have a volume mounted and on line.
SS$_NONEXDRV	Nonexistent drive. The addressed drive does not exist.

(continued on next page)

Table 4-6 (Cont.)
Status Returns for Tape Devices

Status	Meaning
SS$_PARITY	Parity error. One or more of the following conditions can cause this error: CRC error (NRZI only) Control bus parity error Correctable data error (PE only) Correctable skew (PE only) Data bus parity error Incorrectable error (PE only) Invalid tape mark (NRZI only) Nonstandard gap Longitudinal parity error (NRZI only) Format error (PE only) Vertical parity error (NRZI only) Map parity error MASSBUS control parity error MASSBUS data parity error Read data substitute
SS$_UNSAFE	Drive unsafe. The addressed drive is currently unsafe and cannot perform any function.
SS$_VOLINV	Volume invalid. The addressed drive has not been mounted and therefore does not have volume valid set, or a status change has occurred in the drive so that it has changed to an unknown, and therefore, invalid state. All logical and virtual functions will be rejected with this status until volume valid is set. Volume valid is set when a volume is mounted and cleared when the volume is unloaded, the respective drive changes to an unknown state, or the power fails. The driver automatically sets volume valid when the proper volume is mounted and/or power is restored.
SS$_WRITLCK	Write-lock error. An attempt was made to write on a write-locked drive.
SS$_DATAOVERUN	Data overrun. The data block read was longer than the assigned buffer. In the case of a read reverse, the last data on tape (that is, the data nearest the end-of-tape at the beginning of the operation) is the first data read. This data is in the buffer.

4.6 PROGRAMMING EXAMPLE

The following program is an example of how data is written to and read from magnetic tape. In the example, QIO operations are performed through the magnetic tape ACP. These operations could have been performed directly on the device using the magnetic tape driver. However, this would have involved additional programming, for example, writing header labels and trailer labels.

```
        .TITLE MAGTAPE PROGRAMMING EXAMPLE
        .IDENT  /01/

;
; Define necessary symbols
;

        $FIBDEF                         ;Define file information block symbols
        $IODEF                          ;Define I/O function codes

;
; Allocate storage for the necessary data structures
;

;
; Allocate magtape device name string and descriptor
;

TAPENAME:                               ;
        .LONG   20$-10$                 ;Length of name string
        .LONG   10$                     ;Address of name string
10$:    .ASCII  /TAPE/                  ;Name string
20$:                                    ;Reference label

;
; Allocate space to store assigned channel number
;

TAPECHAN:                               ;
        .BLKW   1                       ;Tape channel number

;
; Allocate space for the I/O status quadword
;

IOSTATUS:                               ;
        .BLKQ   1                       ;I/O status quadword

;
; Allocate storage for the input/output buffer
;

BUFFER:
        .REPT   256
        .ASCII  /A/                     ;Initialise buffer to contain 'A'
        .ENDR

; We now define the FIB-file information block-which the ACP uses
; in order to access,deaccess the file.We supply some information
; in this block and the ACP will supply further information.
;

FIB_DESCR:                              ;Start of FIB
        .LONG   ENDFIB-FIB              ;Length of file information block
        .LONG   FIB                     ;Address of file information block
FIB:    .LONG   FIB$M_WRITE!FIB$M_NOWRITE ;Read/write access allowed
        .WORD   0,0,0                   ;File ID
        .WORD   0,0,0                   ;Directory ID
        .LONG   0                       ;Context
        .WORD   0                       ;Name flags
        .WORD   0                       ;Extend control
ENDFIB:                                 ;Reference label

;
; we now define the file name string and descriptor
;

NAME_DESCR:                             ;
        .LONG   END_NAME-NAME           ;File name descriptor
        .LONG   NAME                    ;Address of name string
NAME:   .ASCII  "MYDATA.DAT;1"          ;File name string
END_NAME:                               ;Reference label

;
; Now the main program
;
; The program firstly assigns a channel to the magnetic tape unit.
; It then performs an access function to create and access a file
; called "MYDATA.DAT". It now writes 26 blocks of data to the tape
; containing the letters of the alphabet. The first block contains
; all A's the next all B's and so on. It starts by writing a block
; of 256 bytes and each subsequent block is reduced in size by two
; bytes so by the time it writes the block containing Z's the block
; size is only 206 bytes. The magtape ACP will not allow reading of
; a file that has been written until one of three things happens.
; The file is de-accessed,the file is rewound or the file is back-
; spaced. In this example the file is backspaced zero blocks and
; then it is read in reverse (incrementing the block size every block
; and the data checked against what is meant to be there. If all is
; well the file is de-accessed and the program exits
;
```

MAGNETIC TAPE DRIVER

```
        .ENTRY  MAGTAPE_EXAMPLE,^M<R3,R4,R5,R6,R7,R8>

;
; First assign a channel to the tape unit
;

        $ASSIGN_S TAPENAME,TAPECHAN     ;Assign tape unit
        CMPW    #SS$_NORMAL,R0          ;OK?
        BSBW    ERRCHECK                ;Find out

;
; Next create and access the file 'MYDATA.DAT'

;

        $QIOW_S CHAN=TAPECHAN,-         ;Channel is magtape
                FUNC=#IO$_CREATE!IO$M_ACCESS!IO$M_CREATE,-;Function is create
                IOSB=IOSTATUS,-         ;Address of I/O status word
                P1=FIB_DESCR,-          ;FIB descriptor
                P2=#NAME_DESCR          ;Name descriptor
        CMPW    #SS$_NORMAL,R0          ;OK?
        BSBW    ERRCHECK                ;Find out

;
; LOOP1 consists of writing the alphabet to the tape as described earlier
;

        MOVL    #26,R5                  ;Set up loop count
        MOVL    #256,R3                 ;Set up initial byte count in R3
LOOP1:                                  ;Start of loop
        $QIOW_S CHAN=TAPECHAN,-         ;Perform QIO to tape channel
                FUNC=#IO$_WRITEVBLK,-   ;Function is write virtual block
                P1=BUFFER,-             ;Buffer address
                P2=R3                   ;Byte count
        CMPW    #SS$_NORMAL,R0          ;OK?
        BSBW    ERRCHECK                ;Find out

;
; Now we decrement the byte count ready for the next write, set up a
; loop count for updating the character and LOOP2 performs the update
;

        SUBL2   #2,R3                   ;Decrement byte count for next write
        MOVL    R3,R8                   ;Copy byte count to R8 for LOOP2 count
        MOVAL   BUFFER,R7               ;Get buffer address in R7
LOOP2:  INCB    (R7)+                   ;Increment character
        SOBGTR  R8,LOOP2                ;Until finished
        SOBGTR  R5,LOOP1                ;Repeat LOOP1 until alphabet complete

;
; We now fall through LOOP1 and should update the byte count so that
; it truly reflects the size of the last block written to the tape
;

        ADDL2   #2,R3                   ;Update byte count

;
; We now want to read the tape but must first perform one of the three
; operations outlined above otherwise the ACP will not allow write
; access. We will perform an ACP control function on it specifying
; skip zero blocks. This is a special case of skip reverse and will
; cause the ACP to now allow read access.
;

        CLRL    FIB+FIB$L$_CNTRLVAL     ;Set up to space zero blocks
        MOVW    #FIB$C_SPACE,FIB+FIB$W$_CNTRLFUNC ;Set up for space function
        $QIOW_S CHAN=TAPECHAN,-         ; Perform QIO to tape channel
                FUNC=#IO$_ACPCONTROL,-  ;Perform an ACP control function
                P1=FIB_DESCR            ;Define the FIB
        CMPW    #SS$_NORMAL,R0          ;Success?

        BSBW    ERRCHECK                ;Find out

;
; Now we read the file in reverse
;

        MOVL    #26,R5                  ;Set up loop count
        MOVB    #^A/Z/,R6               ;Get first character in R6
LOOP3:
        MOVAL   BUFFER,R7               ;And buffer address to R7
        $QIOW_S CHAN=TAPECHAN,-         ;Channel is magtape
                FUNC=#IO$_READVBLK!IO$M_REVERSE,- ;Function is read reverse
                IOSB=IOSTATUS,-         ;Define I/O status quadword
                P1=BUFFER,-             ;And buffer address
                P2=R3                   ;R3 bytes
        CMPW    #SS$_NORMAL,R0          ;Success?
        BSBW    ERRCHECK                ;Find out

;
; Now we will check the data we have read in to make sure
; that it agrees with what was written
;

        MOVL    R3,R4                   ;Copy R3 to R4 for loop count
CHECKDATA:                              ;
        CMPB    (R7)+,R6                ;Check each character
        BNEQ    MISMATCH                ;Print message on error
        SOBGTR  R4,CHECKDATA            ;Carry on until finished
        DECB    R6                      ;Go backwards through alphabet
        ADDL2   #2,R3                   ;Update byte count by 2 for next block
        SOBGTR  R5,LOOP3                ;Read next block
```

```
;
; Now we deaccess the file
;

        $QIOW_S CHAN=TAPECHAN,-         ;Channel is magtape
                FUNC=#IO$_DEACCESS,-    ;Deaccess function
                IOSB=IOSTATUS           ;I/O status


;
; Now we deassign the channel and exit
;

EXIT:   $DASSGN_S CHAN=TAPECHAN         ;Deassign channel
        RET                             ;Exit

;
; we are now at a place where normally we would attempt to generate some error
; message but for this example we will simply exit
;

MISMATCH:                               ;
        BRB     EXIT                    ;Exit
ERRCHECK:                               ;If error then exit
        BNEQ    EXIT                    ;Exit if not OK
        RSB                             ;Else return
.END    MAGTAPE_EXAMPLE
```

CHAPTER 5

LINE PRINTER DRIVER

This chapter describes the use of the VAX/VMS line printer driver. This driver supports the LP11 Line Printer Interface and the LA11 DECprinter I.

5.1 SUPPORTED LINE PRINTER DEVICES

The following sections describe the LP11 Line Printer Interface and the LA11 DECprinter I.

5.1.1 LP11 Line Printer Interface

The LP11 is a high-speed, 132-column, line printer available in several models. Printers are available with either a 64- or 96-character ASCII print set. The LP11-R and LP11-S are fully buffered models that operate at a standard speed of 1110 lines per minute. Other LP11 models have 20-character print buffers, and can print at full speed if the printed line is 20 characters or less. Longer lines are printed at a slower rate. Forms with up to six parts can be used for multiple copies.

5.1.2 LA11 DECprinter I

The LA11 DECprinter I is a medium-speed printer that operates at a standard speed of 180 characters per second. It incorporates such features as a forms length switch to set the top of form to any of 11 common lengths, paper-out switch and alarm, and variable forms width. The LA11 uses a 96-character ASCII set; the column width is 132 characters.

5.2 DRIVER FEATURES AND CAPABILITIES

The VAX/VMS line printer driver provides output character formatting and error recovery, as described in the following sections.

5.2.1 Output Character Formatting

In write virtual and write logical block operations, user-supplied characters are output as follows (write physical block data is not formatted, but output directly):

- Rubouts are discarded.
- Tabs move the horizontal print position to the next MODULO (8) position.
- All lowercase alphabetic characters are converted to uppercase before printing (unless the characteristic specifying lowercase characters is set; see Section 5.4.2 and Table 5-2).
- On printers where the line feed, form feed, vertical tab, and return characters empty the printer buffer, returns are held back and output only if the next character is not a form feed, line feed, or vertical tab. Returns are always output on units that have the return function characteristic set (see Section 5.4.3 Table 5-2).
- The horizontal print position is incremented on the output of all nonprinting characters such as the space character. Nonprinting characters are discarded if the horizontal print position is equal to or greater than the carriage width.
- On printers without mechanical form feed (the form feed function characteristic is not set; see Section 5.4.3 and Table 5-2), a form feed is converted to multiple line feeds. The number of line feeds is based on the current line count and the page length.
- Print lines are counted and returned to the caller in the second longword of the I/O status block.

5.2.2 Error Recovery

The VAX/VMS line printer driver performs the following error recovery operations:

- If the printer is offline for 30 seconds, a "device not ready" message is sent to the system operator process.
- If the printer runs out of paper or has a fault condition, a "device not ready" message is sent to the system operator every 30 seconds.
- The current operation is retried every 2 seconds to test for a changed situation, for example, the printer coming online.
- The current I/O operation can be canceled at the next timeout without the printer being online.
- When the printer comes online, device operation resumes automatically.

5.3 DEVICE INFORMATION

The user process can obtain information on printer characteristics by using the $GETCHN and $GETDEV system services (see Section 1.10). The printer-specific information is returned in the first three longwords of a user-specified buffer, as shown in Figure 5-1 (Figure 1-8 shows the entire buffer).

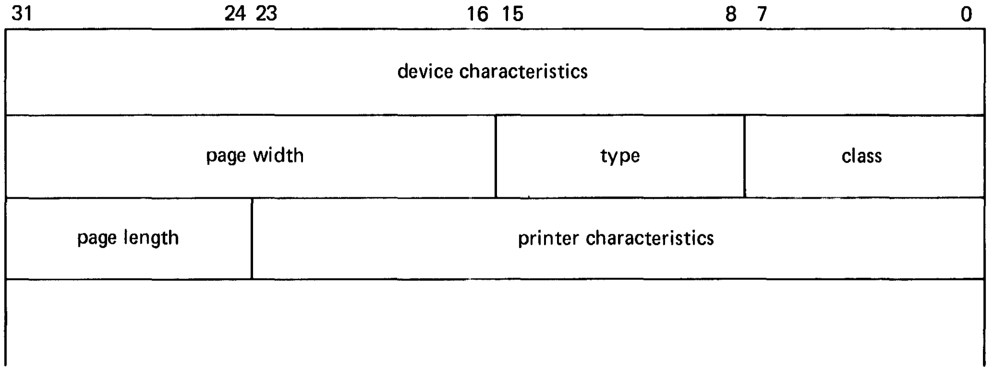


Figure 5-1 Printer Information

The first longword contains device-independent data. The second and third longwords contain device-dependent data.

Table 5-1 lists the device-independent characteristics returned in the first longword.

Table 5-1
Printer Device-Independent Characteristics

Dynamic Bits[1] (Conditionally Set)	Meaning
DEV$M_SPL	Spooled device
DEV$M_AVL	Printer is online and available
Static Bits[1] (Always Set)	
DEV$M_REC	Record-oriented device
DEV$M_CCL	Carriage control
DEV$M_ODV	Device is capable of output

[1] Defined by the $DEVDEF macro.

In the second longword, the device class is DC$_LP. The printer type is a value that corresponds to the printer: LP$_LP11 or LP$_LA11. The page width is a value in the range of 0 to 255.

The third longword contains printer characteristics and the page length. The printer characteristics part can contain any of the values listed in Table 5-2.

Table 5-2
Printer Device-Dependent Characteristics

Value	Meaning
LP$M_LOWER	Printer can print lowercase characters. If this value is not set, all lowercase characters are converted to uppercase when output.
LP$M_MECHFORM	Printer has mechanical form feed. This characteristic is used when variable form length is required, for example, check printing. Driver sends ASCII form feed (decimal 12). Otherwise, multiple line feeds are generated. The page length determines the number of line feeds.
LP$M_CR	Printer requires carriage return. (See note 4, Section 5.2.1).

Maximum page length is 255.

The $LPDEF macro defines the values for the printer characteristics; the $DCDEF macro defines the device class and types.

5.4 LINE PRINTER FUNCTION CODES

The basic line printer I/O functions are write, sense mode, and set mode. None of the function codes takes function modifiers.

5.4.1 Write

The line printer write functions print the contents of the user buffer on the designated printer.

The write functions and their QIO function codes are:

- IO$_WRITEVBLK - write virtual block
- IO$_WRITELBLK - write logical block
- IO$_WRITEPBLK - write physical block (the data is not formatted, but output directly, as in PASSALL mode on terminals)

The write function codes can take the following device/function dependent arguments:

- P1 = the starting virtual address of the buffer that is to be written
- P2 = the number of bytes that are to be written
- P3 (ignored)
- P4 = carriage control specifier except for write physical block operations (write function carriage control is described in Section 5.4.1.1)

P3, P5, and P6 are not meaningful for line printer write operations.

In write virtual block and write logical block operations, the buffer specified by P1 and P2 is formatted for the selected line printer and includes the carriage control information specified by P4.

If the printer is not set spooled, write virtual and write logical perform the same function. If the printer is set spooled, a write logical function queues the I/O to the printer and a write virtual function queues the I/O to the intermediate device, usually a disk.

All lowercase characters are converted to uppercase if the characteristics of the selected terminal do not include LP$M_LOWER (this does not apply to write physical block operations).

Multiple line feeds are generated for form feeds only if the printer does not have a mechanical form feed, that is, the LP$M_MECHFORM characteristic. The number of line feeds generated depends on the current page position and the length of the page.

Section 5.2.1 describes character formatting in greater detail.

5.4.1.1 **Write Function Carriage Control** - The P4 argument is a longword that specifies carriage control. Carriage control determines the next printing position on the line printer. (P4 is ignored in a write physical block operation.) Figure 5-2 shows the P4 longword format.

	3	2	1	0
P4:	POSTFIX	PREFIX	(not used)	FORTRAN

Figure 5-2 P4 Carriage Control Specifier

Only bytes 0, 2, and 3 in the longword are used. Byte 1 is ignored. If the low-order byte (byte 0) is not 0, the contents of the longword are interpreted as a FORTRAN carriage control specifier. Table 5-3 lists the possible byte 0 values (in hexadecimal) and their meanings.

If the low-order byte (byte 0) is 0, bytes 2 and 3 of the P4 longword are interpreted as the prefix and postfix carriage control specifiers. The prefix (byte 2) specifies the carriage control before the buffer contents are printed. The postfix (byte 3) specifies the carriage control after the buffer contents are printed. The sequence is:

Prefix carriage control - Print - Postfix carriage control

The prefix and postfix bytes, although interpreted separately, use the same encoding scheme. Table 5-4 shows this encoding scheme in hexadecimal format.

Table 5-3
Write Function Carriage Control (FORTRAN: Byte 0 not equal to 0)

Byte 0 Value (hexadecimal)	ASCII Character	Meaning
20	(space)	Single-space carriage control. (Sequence: newline, print buffer contents, return.)
30	0	Double-space carriage control. (Sequence: newline, newline, print buffer contents, return.)
31	1	Page eject carriage control. (Sequence: form feed, print buffer contents, return.)
2B	+	Overprint carriage control. (Sequence: print buffer contents, return.) Allows double printing for emphasis or special effects.
24	$	Prompt carriage control. (Sequence: newline, print buffer contents.)
All other values		Same as ASCII space character: single-space carriage control.

Table 5-4
Write Function Carriage Control (P4 byte 0 equal to 0)

Prefix/Postfix Bytes
(Hexadecimal)

Bit 7	Bits 0 - 6	Meaning
0	0	No carriage control is specified, that is, NULL.
0	1-7F	Bits 0 through 6 are a count of newlines (carriage return followed by line feed).

Bit 7	Bit 6	Bit 5	Bits 0-4	Meaning
1	0	0	1-1F	Output the single ASCII control character specified by the configuration of bits 0 through 4 (7-bit character set).
1	1	0	1-1F	Output the single ASCII control character specified by the configuration of bits 0 through 4 which are translated as ASCII characters 128 through 159 (8-bit character set).

Figure 5-3 shows the prefix and postfix hexadecimal coding that produces the carriage control functions listed in Table 5-3. Prefix and postfix coding provides an alternative way to achieve these controls.

(Space)

P4:	8D	1	–	0

Sequence:
Prefix = NL
Print
Postfix = CR

"0"

P4:	8D	2	–	0

Sequence:
Prefix = LF, LF
Print
Postfix = CR

"1"

P4:	8D	8C	–	0

Sequence:
Prefix = FF
Print
Postfix = CR

"+"

P4:	8D	0	–	0

Sequence:
Prefix = NULL
Print
Postfix = CR

"$"

P4:	0	8A	–	0

Sequence:
Prefix = NL
Print
Postfix = NULL

Example: Skip 24 lines before printing

P4:	8D	18	–	0

Sequence:
Prefix = 24NL
Print
Postfix:= CR

Figure 5-3 Write Function Carriage Control
(Prefix and Postfix Coding)

In the first example, the prefix/postfix coding for a single-space carriage control (line feed, print buffer contents, return) is obtained by placing the value (1) in the second (prefix) byte and the sum of the bit 7 value (80) and the return value (D) in the third (postfix) byte:

```
  80 (bit 7 = 1)
+  D (return)
----
  8D (postfix = return)
```

5.4.2 Sense Printer Mode

This function senses the current device-dependent printer characteristics and returns them in the second longword of the I/O status block. No device/function-dependent arguments are used with IO$_SENSEMODE.

5.4.3 Set Mode

Set mode operations affect the operation and characteristics of the associated line printer. VAX/VMS provides two types of set mode functions: Set Mode and Set Characteristics. Set Mode requires logical I/O privilege. Set Characteristics requires physical I/O privilege. Two function codes are provided:

- IO$_SETMODE
- IO$_SETCHAR

These functions take the following device/function-dependent argument (other arguments are not valid):

- P1 -- the address of a characteristics buffer

Figure 5-4 shows the quadword P1 characteristics buffer for IO$_SETMODE. Figure 5-5 shows ths same buffer for IO$_SETCHAR.

31 24	23 16	15 0
page width		not used
page length	printer characteristics	

Figure 5-4 Set Mode Characteristics Buffer

31 24	23 16	15 8	7 0
page width		type	class
page length	printer characteristics		

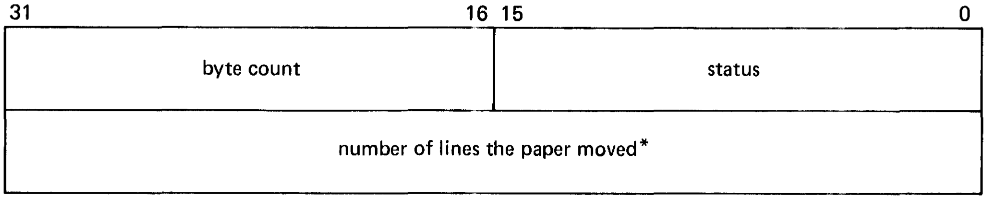
Figure 5-5 Set Characteristic Characteristics Buffer

In the buffer, the device class is DC$_LP. The printer type is a value that corresponds to the printer: DT$_LP11 or DT$_LA11. The type can be changed by the IO$_SETCHAR function. The page width is a value in the range of 0 to 255.

The printer characteristics part of the buffer can contain any of the values listed in Table 5-2.

5.5 I/O STATUS BLOCK

The I/O status blocks (IOSB) for the write and set mode I/O functions are shown in Figures 5-6 and 5-7. Table 5-5 lists the status returns for these functions.



*0 if IO$_WRITEPBLK

Figure 5-6 IOSB Contents - Write Function

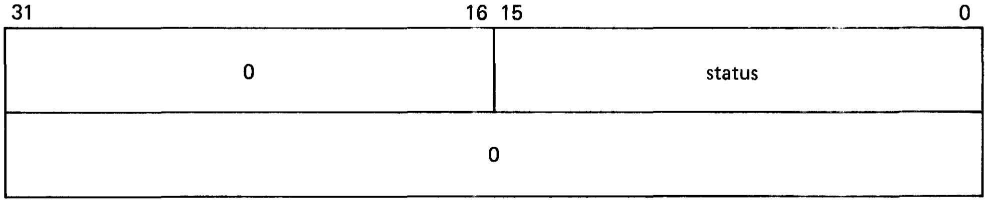


Figure 5-7 IOSB Contents - Set Mode Function

Table 5-5
Line Printer QIO Status Returns

Status	Meaning
SS$_NORMAL	Successful completion. The operation specified in the QIO was completed successfully. On a write operation, the second word of the IOSB can be examined to determine the number of bytes written.
SS$_ABORT	The operation was canceled by the Cancel I/O on Channel ($CANCEL) system service.

5.6 PROGRAMMING EXAMPLE

The following simple program is an example of I/O to the line printer that shows how to use the different carriage control formats. This program prints out the contents of the output buffer (OUT_BUFFER) 10 times using 10 different carriage control formats. The formats are held in location OUTPUT_FORMAT.

```
        .TITLE  LINE PRINTER PROGRAMMING EXAMPLE
        .IDENT  /01/
;Define necessary symbols
;
        $IODEF                          ;Define I/O function codes

;
; Allocate storage for the necessary data structures
;
; Allocate output buffer and fill with required output text
;
OUT_BUFFER:
        .ASCII  "VAX_PRINTER_EXAMPLE"
OUT_BUFFER_SIZE=.-OUT_BUFFER            ;Define size of output string

;
; Allocate device name string and descriptor
;
DEVICE_DESCR:                           ;
        .LONG   20$-10$                 ;Length of name string
        .LONG   10$                     ;Address of name string
10$:    .ASCII  /LINE_PRINTER/          ;Name string of output device
20$:                                    ;Reference label to calculate length

;
; Allocate space to store assigned channel number
;
DEVICE_CHANNEL:                         ;
        .BLKW   1                       ;Channel number

;
; Now set up the carriage control formats
;
OUTPUT_FORMAT:                          ;
        .BYTE   0,0,0,0                 ;No  carriage  control
        .BYTE   32,0,0,0                ;Blank=LF+...TEXT..+CR
        .BYTE   48,0,0,0                ;Zero=LF+LF+.TEXT..+CR
        .BYTE   49,0,0,0                ;One=FF+...TEXT....+CR
        .BYTE   43,0,0,0                ;Plus=Overprint....+CR
        .BYTE   36,0,0,0                ;Dollar=LF+TEXT(Prompt)
;
; Now the prefix-postfix carriage control formats
;
        .BYTE   0,0,1,141               ;LF+.....TEXT.....+CR

        .BYTE   0,0,24,141              ;24LF+...TEXT.....+CR
        .BYTE   0,0,2,141               ;LF+LF+..TEXT.....+CR
        .BYTE   0,0,140,141             ;FF+.....TEXT.....+CR
; Program starting point
;
; The program assigns a channel to the output device,sets up a loop
; Count for the number of times it wishes to print, and performs ten
; QIO and wait system services.The channel is then deassigned.
;
        .ENTRY  PRINTER_EXAMPLE,^M<R2,R3>;Program starting address

;
; First assign a channel to the output device
;
        $ASSIGN_S DEVNAM=DEVICE_DESCR,- ;Assign a channel to printer
                CHAN=DEVICE_CHANNEL     ;
        BLBC    R0,50$                  ;If low bit clear,assignment failure
        MOVL    #11,R3                  ;Set up loop count
        MOVAL   OUTPUT_FORMAT,R2        ;Set up o/p format address in R2

;
; Start of printing loop
;
30$:    $QIOW_S CHAN=DEVICE_CHANNEL,-   ;Print on device channel
                FUNC=#IO$_WRITEVBLK,-   ;I/O function is write virtual
                P1=OUT_BUFFER,-         ;Address of output buffer
                P2=#OUT_BUFFER_SIZE,-   ;Size of buffer to print
                P4=(R2)+                ;Format control in R2
                                        ;Will auto-increment.
        BLBC    R0,40$                  ;If low bit clear,i/o failure
        SOBGTR  R3,30$                  ;Branch if not finished
40$:    $DASSGN_S CHAN=DEVICE_CHANNEL   ;Deassign channel
50$:    RET                             ;Return

        .END    PRINTER_EXAMPLE
```

CHAPTER 6

CARD READER DRIVER

This chapter describes the use of the VAX/VMS card reader driver. This driver supports the CR11 Card Reader.

6.1 SUPPORTED CARD READER DEVICE

The CR11 Card Reader reads standard 80-column punched data cards.

6.2 DRIVER FEATURES AND CAPABILITIES

The VAX/VMS card reader driver provides the following capabilities:

- Multiple controllers of the same type; for example, more than one CR11 can be used on the system
- Binary, packed Hollerith, and translated 026 or 029 read modes
- Unsolicited interrupt support for automatic card reader input spooling
- Special card punch combinations to indicate an end-of-file condition and to set the translation mode
- Error recovery

The following sections describe the read modes, special card punch combinations, and error recovery in greater detail.

6.2.1 Read Modes

VAX/VMS provides two card reader device/function-dependent modifier bits for read data operations: read packed Hollerith (IO$M_PACKED) and read binary (IO$M_BINARY). If IO$M_PACKED is set, the data is packed and stored in sequential bytes of the input buffer. If IO$M_BINARY is set, the data is read and stored in sequential words of the input buffer. IO$M_BINARY takes precedence over IO$M_PACKED.

The read mode can also be set by a set translation mode card (see Section 6.2.2.2) or by the Set Mode function (see Section 6.4.3).

6.2.2 Special Card Punch Combinations

The VAX/VMS card reader driver recognizes three special card punch combinations in column 1 of a card. One combination signals an end-of-file condition. The other two combinations set the current translation mode.

6.2.2.1 **End-of-File Condition** - A card with the 12-11-0-1-6-7-8-9 holes punched in column 1 signals an end-of-file condition. If the read mode is binary, the first eight columns must contain this punch combination.

6.2.2.2 **Set Translation Mode** - If the read mode is nonbinary, nonpacked Hollerith (the IO$M_BINARY and IO$M_PACKED function modifiers are not set), the current translation mode can be set to the 026 or 029 punch code. A card with the 12-2-4-8 holes punched in column 1 sets the translation mode to the 026 code. A card with the 12-0-2-4-6-8 holes punched in column 1 sets the translation mode to the 029 code. The translation mode can be changed as often as required.

If a translation mode card contains punched information in columns 2 through 80, it is ignored.

Logical, virtual, and physical read functions result in only one card's being read. If a translation mode card is read, the read function is not completed and another card is read immediately.

6.2.3 Error Recovery

The VAX/VMS card reader driver performs the following error recovery operations:

- If the card reader is offline for 30 seconds, a "device not ready" message is sent to the system operator.
- If a recoverable card reader failure is detected, a "device not ready" message is sent to the system operator every 30 seconds.
- The current operation is retried every two seconds to test for a changed situation, for example, the removal of an error condition.
- The current I/O operation can be canceled at the next timeout without the card reader being online. When the card reader comes online, device operation resumes automatically.

There are four categories of card reader failures:

- Pick check -- The next card cannot be delivered from the input hopper to the read mechanism.
- Stack check -- The card just read did not stack properly in the output hopper.

- Hopper check -- Either the output hopper is full or the input hopper is empty.

- Read check -- The last card was read incorrectly due to torn edges or punches before column 1 or after column 80.

Manual intervention is required if any of these errors occur. The recovery is transparent to the user program issuing the I/O request.

When a recoverable card reader failure is detected, a "device not ready" message is displayed on the system operator console. When this message is received, the card reader indicator lights should be examined to determine the reason for the failure. The indicator lights and the respective recovery procedures are:

- Pick check -- The next card cannot be delivered to the read mechanism. Remove the next card to be read from the input hopper and smooth the leading edge, that is, the edge that will enter the read mechanism first. Replace the card in the input hopper and press the RESET button. Card reader operation will resume automatically. If a pick check error occurs again on the same card, remove the card from the input hopper and repunch it. Place the duplicate card in the input hopper and press the RESET button. If the problem persists, either an adjustment is required or nonstandard cards are in the input hopper.

- Stack check -- The card just read did not stack properly in the output hopper. Remove the last card read from the output hopper and examine the condition. If it is excessively worn or mutilated, repunch it. Place either the duplicate or the original card in the read station of the input hopper and press the RESET button. Card reader operation will resume automatically. If the stack check error recurs immediately, an adjustment is required.

- Hopper check -- Either the input hopper is empty or the output hopper is full. Examine the input hopper and, if empty, either load the next deck of input cards or an end of file card. If the input hopper is not empty, remove the cards that have accumulated in the output hopper and press the RESET button. Card reader operation will resume automatically.

- Read check -- The last card was read incorrectly. Remove the last card from the output hopper and examine its condition. If it is excessively worn, mutilated, or contains punches before column 0 or after column 80, repunch the card correcting any incorrect punches. Place either the original or duplicate card in the read station of the input hopper and press the RESET button. Card reader operation will resume automatically. If the read check error recurs immediately, an adjustment is necessary.

6.3 DEVICE INFORMATION

Users can obtain information on card reader characteristics by using the $GETCHN and $GETDEV system services (see Section 1.10). The information is returned in a user-specified buffer shown in Figure 6-1. Only the first three longwords of the buffer are shown in Figure 6-1 (Figure 1-9 shows the entire buffer).

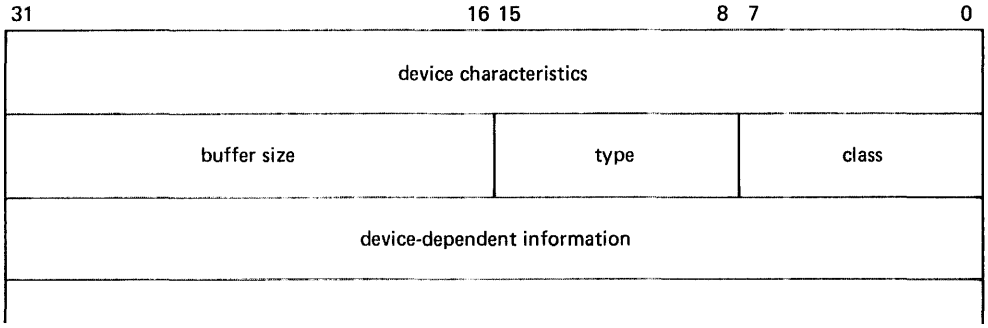


Figure 6-1 Card Reader Information

The device characteristics returned in the first longword are listed in Table 6-1.

Table 6-1
Card Reader Device-Independent Characteristics

Dynamic Bit[1] (Conditionally Set)	Meaning
DEV$M_AVL	Device is online and available
Static Bits[1] (Always Set)	
DEV$M_IDV	Device is capable of input
DEV$M_REC	Device is record oriented

1. Defined by the $DEVDEF macro

The second longword contains information on device class and type, and the buffer size. The device class for card readers is DC$_CARD. The device type is DT$_CR11 for the CR11.

The $DCDEF macro defines the device type and class names. The buffer size is the default to be used for all card reader devices (this default is 80 bytes).

The third longword contains device-dependent card reader characteristics. Table 6-2 lists these characteristics. The $CRDEF macro defines the characterstics values.

Table 6-2
Device-Dependent Information for Card Readers

Value	Meaning
CR$V_TMODE CR$S_TMODE	Specifies the translation mode for nonbinary, nonpacked Hollerith data transfers.[1] Possible values are: CR$K_T026 Translate according to 026 punch code CR$K_T029 Translate according to 029 punch code

[1] Section 6.2.2.2 describes the set translation mode punch code.

6.4 CARD READER FUNCTION CODES

The VAX/VMS card reader can perform logical, virtual, and physical I/O functions. Table 6-3 lists these functions and their function codes. These functions are described in more detail in the following paragraphs.

Table 6-3
Card Reader I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_READLBLK P1,P2	L	IO$M_BINARY IO$M_PACKED	Read logical block
IO$_READVBLK P1,P2	V	IO$M_BINARY IO$M_PACKED	Read virtual block
IO$_READPBLK P1,P2	P	IO$M_BINARY IO$M_PACKED	Read physical block
IO$_SENSEMODE	L		Sense the card reader characteristics and return them in the I/O status block
IO$_SETMODE P1	L		Set card reader characteristics for subsequent operations
IO$_SETCHAR P1	P		Set card reader characteristics for subsequent operations

[1] V = virtual; L = logical; P = physical

6.4.1 Read

This function reads data from the next card in the card reader input hopper into the designated memory buffer in the specified format. Only one card is read each time a read function is specified.

VAX/VMS provides three read function codes:

- IO$_READVBLK - read virtual block
- IO$_READLBLK - read logical block
- IO$_READPBLK - read physical block

Two function-dependent arguments are used with these codes:

- P1 -- the starting virtual address of the buffer that is to receive the data
- P2 -- the number of bytes that are to be read in the specified format

The read binary function modifier (IO$M_BINARY) and the read packed Hollerith function modifier (IO$M_PACKED) can be used with all read functions. If IO$M_BINARY is specified, successive columns of data are stored in sequential word locations of the input buffer. If IO$M_PACKED is specified, successive columns of data are packed and stored in sequential byte locations of the input buffer. If neither of these function modifiers is specified, successive columns of data are translated in the current mode (026 or 029) and stored in sequential bytes of the input buffer. Figure 6-2 shows how data is stored by IO$M_BINARY and IO$M_PACKED.

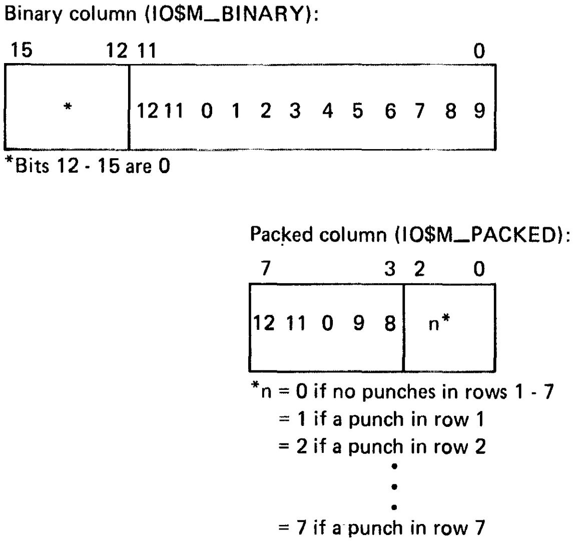


Figure 6-2 Binary and Packed Column Storage

Regardless of the byte count specified by the P2 argument, a maximum of 160 bytes of data for binary read operations and 80 bytes of data for nonbinary read operations (IO$M_PACKED, or 026 or 029 modes) are transferred to the input buffer. If P2 specifies less than the maximum quantity for the respective mode, only the number of bytes specified are transferred; any remaining buffer locations are not filled with data.

6.4.2 **Sense Card Reader Mode**

This function senses the current device-dependent card reader characteristics and returns them in the second longword of the I/O status block (see Table 6-2). No device/function dependent arguments are used with IO$_SENSEMODE.

6.4.3 **Set Mode**

Set mode operations affect the operation and characteristics of the associated card reader device. VAX/VMS defines two types of set mode functions:

- Set Mode
- Set Characteristic

6.4.3.1 **Set Mode** - The Set Mode function affects the characteristics of the associated card reader. Set Mode is a logical I/O function and requires the access privilege necessary to perform logical I/O. A single function code is provided:

IO$_SETMODE

This function takes the following device/function dependent argument:

P1 -- the address of a characteristics buffer

Figure 6-3 shows the quadword Set Mode characteristics buffer.

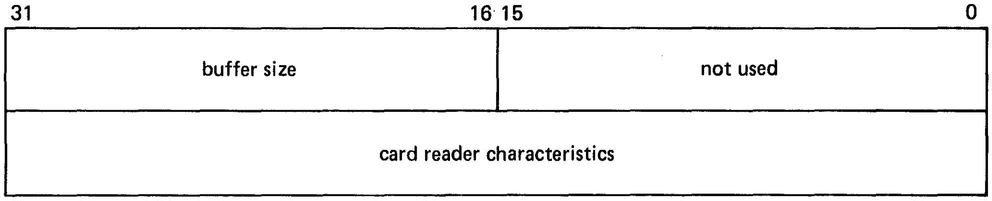


Figure 6-3 Set Mode Characteristics Buffer

Table 6-4 lists the card reader characteristics and their meanings. The $CRDEF macro defines the characteristics values.

Table 6-4
Set Mode and Set Characteristic Card Reader Characteristicss

Value[1]	Meaning
CR$V_TMODE CR$S_TMODE	Specifies the translation mode for nonbinary, nonpacked Hollerith data transfers. Possible values are: CR$K_T026 Translate according to 026 punch code CR$K_T029 Translate according to 029 punch code

[1] If neither the 026 or 029 mode is specified, the default mode can be set by the SET CARD_READER command.

6.4.3.2 **Set Characteristic** - The Set Characteristic function also affects the characteristics of the associated card reader device. Set Characteristic is a physical I/O function and requires the access privilege necessary to perform physical I/O functions. A single function code is provided:

IO$_SETCHAR

This function takes the following device/function dependent argument:

P1 -- the address of a characteristics buffer

Figure 6-4 shows the Set Characteristic characteristics buffer.

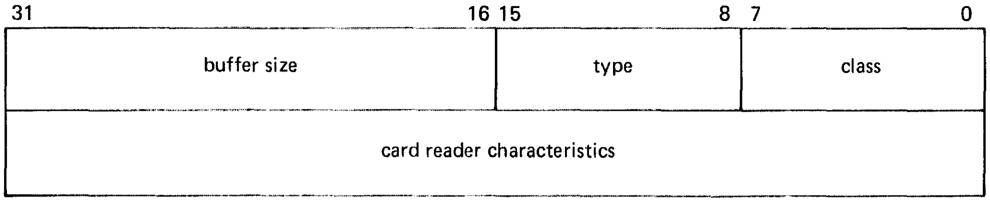


Figure 6-4 Set Characteristic Buffer

The device type value is DT$_CR11. The device class value is DC$_CARD. Table 6-4 lists the card reader characteristics for the Set Characteristic function.

6.5 I/O STATUS BLOCK

The I/O status block (IOSB) format for QIO functions on the card reader is shown in Figure 6-5. Table 6-5 lists the status returns for these functions. Table 6-2 lists the device-dependent data returned in the second longword. The IO$_SENSEMODE function can be used to obtain this data.

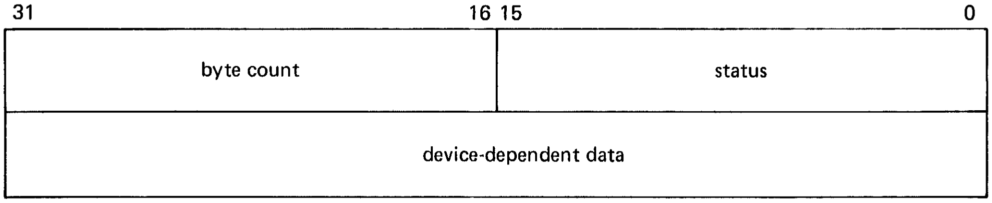


Figure 6-5 IOSB Contents

Table 6-5
Status Returns for Card Reader

Status	Meaning
SS$_NORMAL	Successful completion of the operation specified in the QIO request. The second word of the IOSB can be examined to determine the actual number of bytes written to the buffer.
SS$_DATAOVERRUN	Data overrun. Column data was delivered to the controller data buffer before previous data had been read by the driver.
SS$_ENDOFFILE	End-of-file condition. An end-of-file card was encountered during the read operation.

CHAPTER 7

MAILBOX DRIVER

VAX/VMS supports a virtual device, called a mailbox, that is used for communication between processes. Mailboxes provide a controlled and synchronized method for processes to exchange data. Although mailboxes transfer information in much the same way that other I/O devices do, they are not actual devices. Rather, mailboxes are software-implemented devices that can perform read and write operations.

Multiport memory mailboxes function the same as regular mailboxes. However, they can also be used by processes on different processors that are connected to an MA780.

The VAX/VMS Real-Time User's Guide contains additional information on the use of mailboxes.

7.1 MAILBOX OPERATIONS

Software mailboxes can be compared to the actual metal boxes used for mail delivery. As shown in Table 7-1, both types of mailboxes perform similar operations.

Table 7-1
Mailbox Read and Write Operations

Operation	Use of Conventional Mailboxes	Use of VAX/VMS Software Mailboxes
Receive Mail	Resident checks mailbox to see if any mail was delivered. If so, picks it up, opens it, and reads it.	A process initiates a read to a mailbox to obtain data sent by another process. The process reads data if a message was previously transmitted to the mailbox.
Receive Notification of Mail	The mail carrier leaves notification to the resident that mail can be picked up at the post office.	A process specifies that it wants to be notified through an AST when a message is sent to the mailbox.

(continued on next page)

Table 7-1 (Cont.)
Mailbox Read and Write Operations

Operation	Use of Conventional Mailboxes	Use of VAX/VMS Software Mailboxes
Send Mail (without notification of receipt)	The resident leaves mail addressed to another person in the mailbox, but neither waits for nor expects notification of its delivery.	A process initiates a write request to a mailbox to transmit data to another process. The sending process does not wait until the data is read by the receiving process before completing the I/O operation.
Send Mail (with notification of receipt)	The resident leaves mail addressed to another person in the mailbox and asks to be notified of its delivery.	A process initiates a write request to a mailbox to transmit data to another process. The sending process waits until the receiving process reads the data before completing the I/O operation.
Reject Mail	The resident discards junk mail.	The receiving process reads messages from the mailbox, sorts out unwanted messages, and responds only to useful messages.

7.1.1 Creating Mailboxes

A process uses the Create Mailbox and Assign Channel ($CREMBX) system service to create a mailbox and assign a channel and logical name to it. The system enters the logical name in either the system (permanent mailbox) or group (temporary mailbox) logical name table and gives it an equivalence name of MBAn, where n is a unique unit number.

$CREMBX also establishes the characteristics of the mailbox. These characteristics include a protection mask, permanence indicator, maximum message size, and buffer quota.

Other processes can assign additional channels to the mailbox using either $CREMBX or the Assign I/O Channel ($ASSIGN) system service. The mailbox is identified by its logical name both when it is created and when it is assigned channels by cooperating processes.

Figure 7-1 illustrates the use of $CREMBX and $ASSIGN.

Creating mailboxes requires privilege. If sufficient dynamic memory for the mailbox data structure is not available, a resource wait will occur if resource wait mode is enabled.

The programming example at the end of this chapter (Section 7.5) illustrates mailbox creation and interprocess communication.

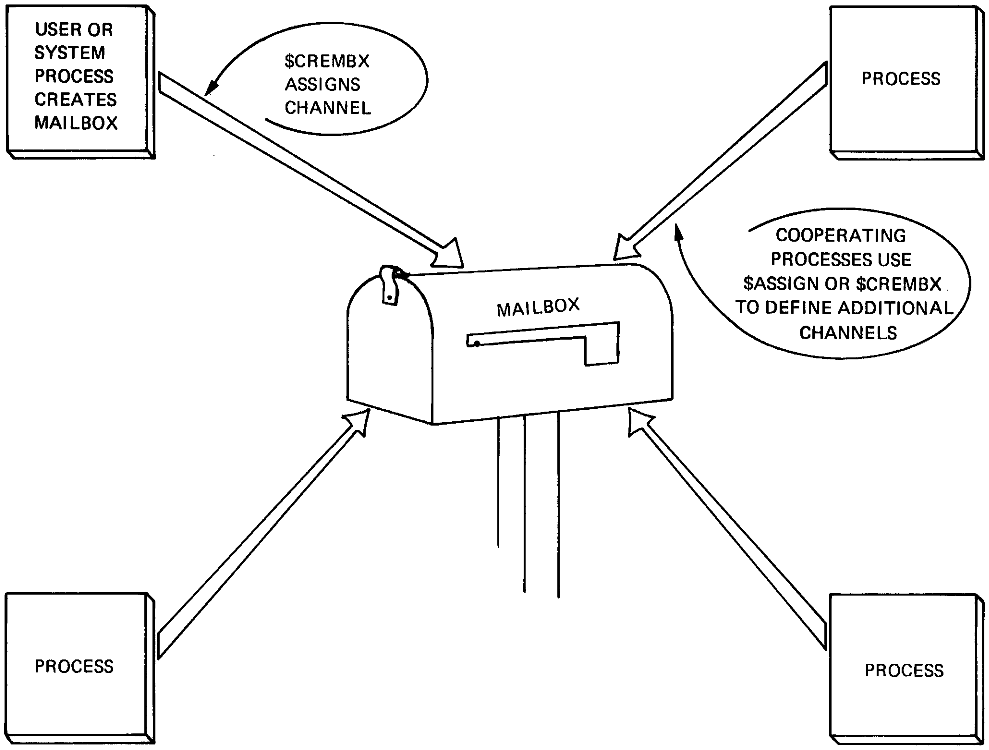

USER OR SYSTEM PROCESS CREATES MAILBOX
$CREMBX ASSIGNS CHANNEL
PROCESS
COOPERATING PROCESSES USE $ASSIGN OR $CREMBX TO DEFINE ADDITIONAL CHANNELS
MAILBOX
PROCESS
PROCESS

Figure 7-1 Multiple Mailbox Channels

7.1.2 Deleting Mailboxes

The system maintains a count of all channels assigned to a temporary mailbox. As each process finishes using a mailbox, it deassigns the channel using the Deassign I/O Channel ($DASSGN) system service. The channel count is decremented by one. The system automatically deletes the mailbox when no more channels are assigned to it (that is, when the channel count reaches 0).

Permanent mailboxes must be explicitly deleted using the Delete Mailbox ($DELMBX) system service. This can occur at any time. However, the mailbox is actually deleted when no processes have channels assigned to it.

When a mailbox is deleted, its message buffer quota is returned to the process that created it.

7.1.3 Mailbox Message Format

There is no standardized format for mailbox messages and none is imposed on users. Figure 7-2 shows a typical mailbox message format. Other types of messages can take different formats; for an example, see Figure 2-1 in Section 2.2.5.

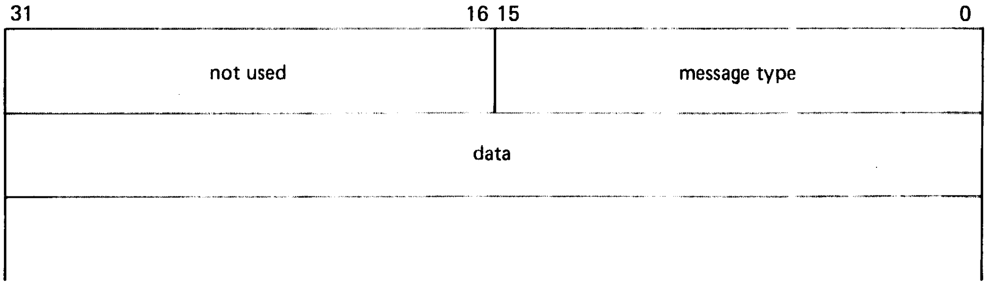


Figure 7-2 Typical Mailbox Message Format

7.2 DEVICE INFORMATION

Users can obtain information on mailbox characteristics by using the $GETCHN and $GETDEV system services (see Section 1.10). The information is returned in a user-specified buffer. The first three longwords of the buffer are shown in Figure 7-3 (Figure 1-9 shows the entire buffer).

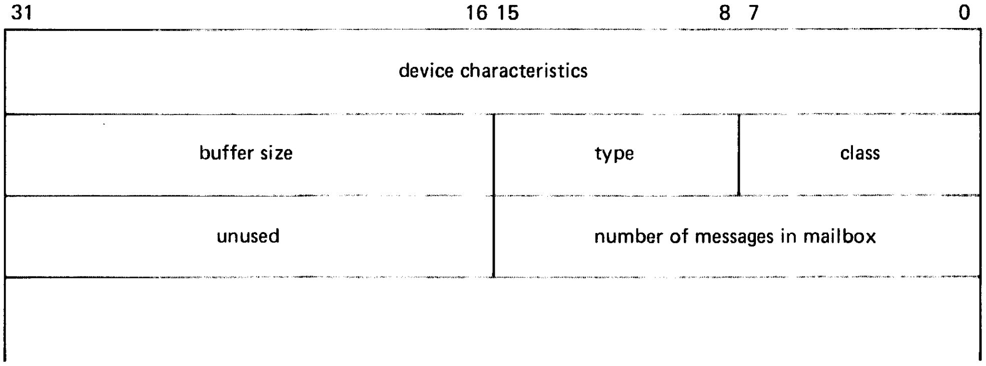


Figure 7-3 Mailbox Information

The first longword in the buffer contains the device characteristics values listed in Table 7-2. The $DEVDEF macro defines these values.

Table 7-2
Mailbox Characteristics

Dynamic Bit (Conditionally Set)	Meaning
DEV$M_SHR	Shareable device
DEV$M_AVL	Device is available
Static Bits (Always Set)	
DEV$M_REC	Record-oriented device
DEV$M_IDV	Device is capable of input
DEV$M_ODV	Device is capable of output
DEV$M_MBX	Mailbox device

The second longword of the buffer contains information on the device class and type, and the buffer size. The device class is DC$_MAILBOX The device type is DT$_MBX. The $DCDEF macro defines these symbols. The buffer size is the maximum message size in bytes.

7.3 MAILBOX FUNCTION CODES

The VAX/VMS mailbox I/O functions are: read, write, write end-of-file, and set attention AST.

No buffered I/O byte count quota checking is performed on mailbox I/O messages. Instead, the byte count or buffer quota of the mailbox is checked for sufficient space to buffer the message being sent. The buffered I/O quota and AST quota are also checked.

7.3.1 Read

Read mailbox QIO requests are used to obtain messages written by other processes. The three mailbox functions and their codes are:

- IO$_READVBLK - read virtual block
- IO$_READLBLK - read logical block
- IO$_READPBLK - read physical block

These function codes take two device/function-dependent arguments:

- P1 -- the starting virtual address of the buffer that is to receive the message read
- P2 -- the size of the buffer in bytes (limited by the maximum message size for the mailbox)

One function modifier can be specified with a QIO read request:

IO$M_NOW -- the I/O operation is completed immediately with no wait for a write request from another process

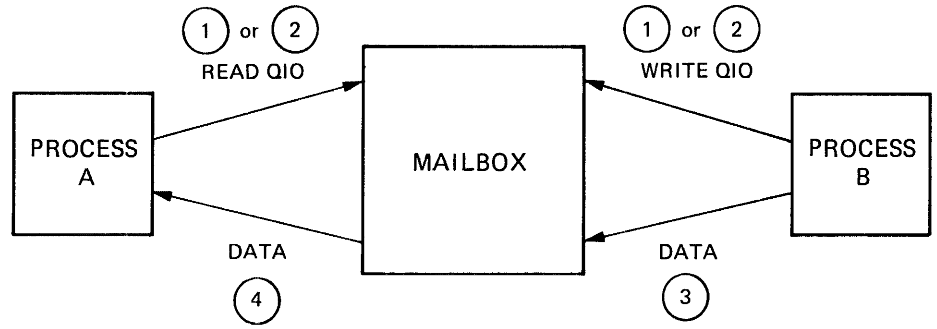
Figure 7-4 illustrates the read mailbox functions; in this figure, Process A reads a mailbox message written by Process B. As the figure indicates, a mailbox read request requires a corresponding mailbox write request (except in the case of an error). The requests can be made in any sequence; that is, the read request can either precede or follow the write request.

Two possibilities exist if Process A issues a read request before Process B issues a write request. If Process A did not specify the function modifier IO$M_NOW, Process A's request is queued during the wait for Process B to issue the write request. When this request occurs, the data is transferred from Process B, through the system buffers, to Process A to complete the I/O operation.

However, if Process A did specify the IO$M_NOW function modifier, the read operation is completed immediately. That is, Process A's request is not queued during the wait for the message from Process B, and no data is transferred from Process B to Process A.

If Process B sends a message (with no function modifier; see Section 7.3.2) before Process A issues a read request (with or without a function modifier), Process A finds a waiting message in the mailbox. The data is transferred and the I/O operation is completed immediately.

To issue the read request, Process A can specify any of the read QIO function codes; all perform the same operation.



NOTE: Numbers indicate order of events.

Figure 7-4 Read Mailbox

7.3.2 Write

Write mailbox QIO requests are used to transfer data from a process to a mailbox. The three mailbox functions and their QIO function codes are:

- IO$_WRITEVBLK -- write virtual block
- IO$_WRITELBLK -- write logical block
- IO$_WRITEPBLK -- write physical block

These function codes take two device/function-dependent arguments:

- P1 -- the starting virtual address of the buffer that contains the message being written
- P2 -- the size of the buffer in bytes (limited by the maximum message size for the mailbox)

One function modifier can be specified with a QIO write request:

IO$M_NOW - the I/O operation is completed immediately with no wait for another process to read the mailbox message

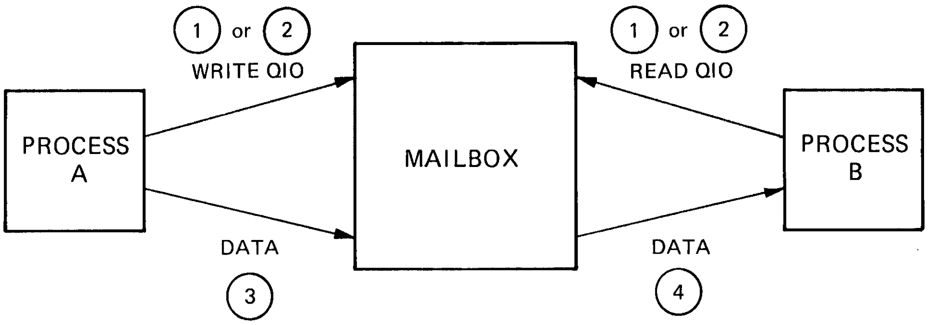
Figure 7-5 illustrates the write mailbox function; in this figure, Process A writes a message to be read by Process B. As in the read request example above, a mailbox write request requires a corresponding mailbox read request (unless an error occurs), and the requests can be made in any sequence.

Two possibilities exist if Process A issues a write request before Process B issues a read request. If Process A did not specify the function modifier IO$M_NOW, Process A's write request is queued during the wait for Process B to issue a read request. When this request occurs, the data is transferred from Process A to Process B to complete the I/O operation.

However, if Process A did specify the IO$M_NOW function modifier, the write operation is completed immediately. The data is available to Process B and is transferred when Process B issues a read request.

If Process B issues a read request (with no function modifier) before Process A issues a write request (with or without the function modifier), Process A finds a waiting request in the mailbox. The data is transferred and the I/O operation is completed immediately.

To issue the write request, Process A can specify any of the write QIO function codes; all perform the same operation.



NOTE: Numbers indicate order of events.

Figure 7-5 Write Mailbox

7.3.3 Write End-of-File Message

Write End-of-File Message QIO requests are used to insert a special message in the mailbox. The process that reads the end-of-file message is returned the status code SS$_ENDOFFILE in the I/O status block. No data is transferred. This function takes no arguments or function modifiers. VAX/VMS provides a single function code:

IO$_WRITEOF -- write end-of-file message

7.3.4 Set Attention AST

Set Attention AST QIO requests are used to specify that an AST be given to notify the requesting process when a cooperating process places an unsolicited read or write request in a designated mailbox. Because the AST only occurs when the read or write request arrives from a cooperating process, the requesting process need not repeatedly check the mailbox status.

The Set Attention AST functions and their function codes are:

- IO$_SETMODE!IO$M_READATTN - read attention AST
- IO$_SETMODE!IO$M_WRTATTN - write attention AST

These function codes take two device/function-dependent arguments:

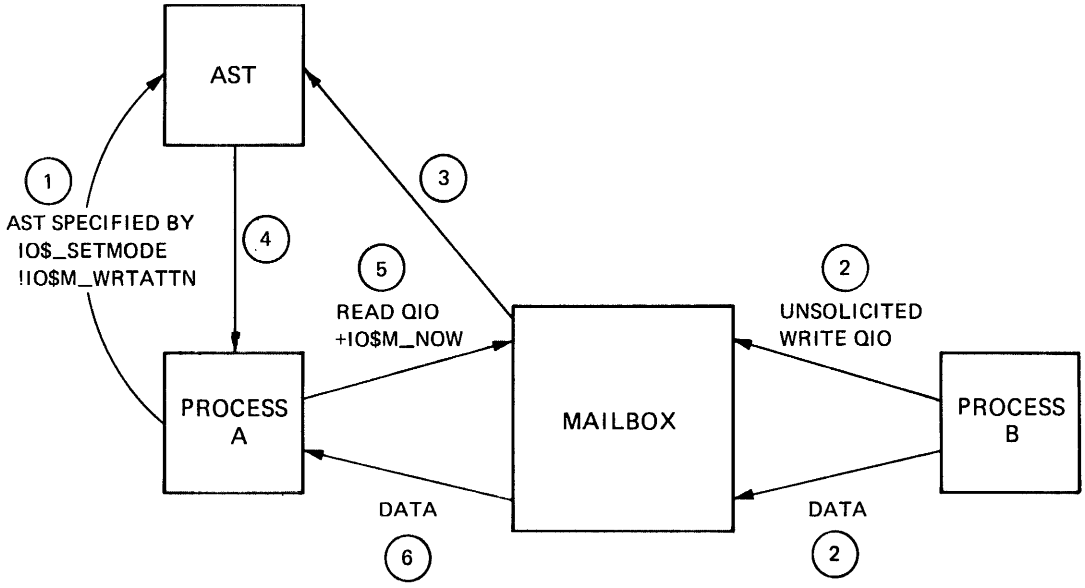
- P1 -- AST address (request notification is disabled if the address is 0)
- P2 -- AST parameter returned in the argument list when the AST service routine is called
- P3 -- access mode to deliver AST; maximized with requester's mode

These functions are one-time AST enables; they must be explicitly reenabled once the AST has been delivered if the user desires notification of the next unsolicited request. Both types of enables, and more than one of each type, can be set at the same time. The number of enables is limited only by the AST quota for the process.

Figure 7-6 illustrates the write attention AST function. In this figure, an AST is set to notify Process A when Process B sends an unsolicited message.

Process A uses the IO$_SETMODE!IO$M_WRTATTN function to request an AST. When Process B sends a message to the mailbox, the AST is delivered to Process A. Process A responds to the AST by issuing a read request to the mailbox. The function modifier IO$M_NOW is included in the read request. The data is then transferred to complete the I/O operation.

If several requesting processes have set ASTs for unsolicited messages at the same mailbox, all ASTs are delivered when the first unsolicited message is placed in the mailbox. However, only the first process to respond to the AST with a read request receives the data. Thus, when the next process to respond to an AST issues a read request to the mailbox, it may find the mailbox empty. If this request does not include the function modifier IO$M_NOW, it will be queued during the wait for the next message to arrive in the mailbox.



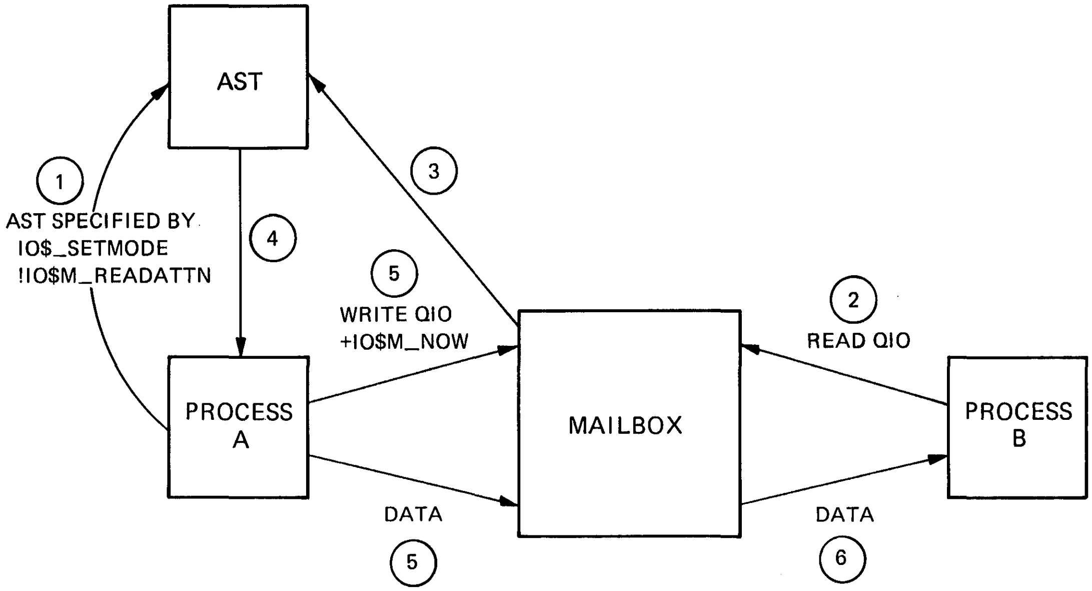
NOTE: Numbers indicate order of events.

Figure 7-6 Write Attention AST (Read Unsolicited Data)

Figure 7-7 illustrates the read attention AST function. In this figure, an AST is set to notify Process A when Process B issues a read request for which no message is available.

Process A uses the IO$_SETMODE!IO$M_READATTN function to specify an AST. When Process B issues a read request to the mailbox, the AST is delivered to Process A. Process A responds to the AST by sending a message to the mailbox. The data is then transferred to complete the I/O operation.

If several requesting processes have set ASTs for read requests at the same mailbox, all ASTs are delivered when the first read request is placed in the mailbox. Only the first process to respond with a write request is able to transfer data to Process B.

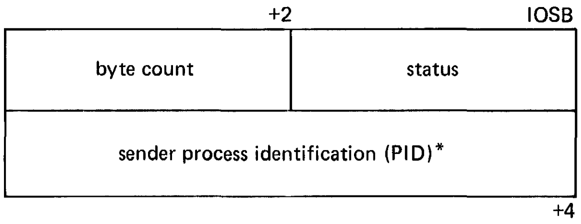


NOTE: Numbers indicate order of events.

Figure 7-7 Read Attention AST

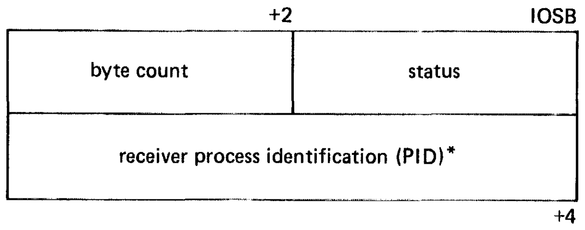
7.4 I/O STATUS BLOCK

The I/O status blocks (IOSB) for mailbox read and write QIO functions are shown in Figures 7-8 and 7-9. Table 7-3 lists the status returns for these functions.



*0 if the sender was a system process

Figure 7-8 IOSB Contents - Read Function



*0 if IO$M_NOW was specified

Figure 7-9 IOSB Contents - Write Function

Table 7-3
Mailbox QIO Status Returns

Status	Meaning
SS$_NORMAL	Successful completion. The operation specified in the QIO was completed successfully. The second word of the IOSB can be examined to determine the number of bytes transferred.
SS$_ENDOFFILE	No message available at the mailbox or end-of-file (IO$_ENDOFFILE) message read.
SS$_NOPRIV	Access denied. The requesting process does not have the privilege to read or write to this mailbox. (The protection mask is defined when the mailbox is created.)
SS$_ACCVIO	Buffer access violation. User buffer is not accessible to the requesting process.
SS$_MBTOOSML	Mailbox too small. The request is for a message that will not fit in the mailbox. Maximum message size is established when the mailbox is created.
SS$_MBFULL	Mailbox full. The mailbox is full and resource wait mode is not enabled.
SS$_INSFMEM	Insufficient dynamic memory for the request. Resource wait mode is not enabled.

7.5 PROGRAMMING EXAMPLE

The following program creates a mailbox and puts some mail in it; no matching read is pending on the mailbox. First, the program illustrates that if the function modifier IO$M_NOW is not used when mail is deposited, the write function will wait until a read operation is performed on the mailbox. In this case, IO$M_NOW is specified and the program continues after the mail is left in the mailbox.

Next, the mailbox is read. If there was no mail in the mailbox the program would wait at this point because IO$M_NOW is not specified. IO$M_NOW should be specified if there is any doubt concerning the availability of data in the mailbox and it is important for the program not to wait.

It is up to the user to coordinate what data goes into and out of mailboxes. In this example the process reads its own message. Normally, two mailboxes are used for interprocess communication: one for sending data from process A to process B, and one for sending data from process B to process A. If a program is arranged in this manner, there is no possibility of a process reading its own message.

```
        MAILBOX DRIVER PROGRAMMING EXAMPLE
        /01/

;
; Define necessary symbols
;

        $IODEF                          ;Define I/O function codes

;
; Allocate storage for necessary data structures
;

;
; Allocate terminal device name string and descriptor
;

DEVICE_DESCR:                           ;
        .LONG   20$-10$                 ;Length of name string
        .LONG   10$                     ;Address of name string
10$:    .ASCII  /TERMINAL/              ;Name string of output device
20$:                                    ;Reference label

;
; Allocate space to store assigned channel number
;

DEVICE_CHANNEL:                         ;
        .BLKW   1                       ;Channel number

;
; Allocate mailbox name string and descriptor
;

MAILBOX_NAME:                           ;
        .LONG   ENDBOX-NAMEBOX          ;Length of name string
        .LONG   NAMEBOX                 ;Address of name string
NAMEBOX:.ASCII  /146_MAIN_ST/           ;Name string

ENDBOX:                                 ;Reference label

;
; Allocate space to store assigned channel number
;

MAILBOX_CHANNEL:                        ;
        .BLKW   1                       ;Channel number

;
; Now allocate space to store the outgoing and incoming messages
;
IN_BOX_BUFFER:                          ;
        .BLKB   40                      ;Allocate 40 bytes for received message
        IN_LENGTH=.-IN_BOX_BUFFER       ;Define input buffer length

OUT_BOX_BUFFER:                         ;
        .ASCII  /SHEEP ARE VERY DIM/    ;Message to send

        OUT_LENGTH=.-OUT_BOX_BUFFER     ;Define length of message to send

;
; Now allocate space for the I/O status quadword
;

STATUS: .QUAD   1                       ;I/O status quadword

;
; Now the program. A mailbox is created and a channel is assigned
; to the terminal. A message is put in the mailbox and a message
; is received from the mailbox (the same message).The contents of
; the mailbox are then printed on the terminal.
;

START:  .WORD   0                       ;Entry mask
        $CREMBX_S CHAN=MAILBOX_CHANNEL,-;Channel is the mailbox
                PROMSK=#^X0000,-        ;No protection
                BUFQUO=#^X0060,-        ;Buffer quota is hex 60
                LOGNAM=MAILBOX_NAME,-   ;Logical name descriptor
                MAXMSG=#^X0060          ;Maximum message is hex 60
        CMPW    #SS$_NORMAL,R0          ;Test for normal return
        BSBW    ERROR_CHECK             ;See if all well
        $ASSIGN_S                      -;Assign channel
                DEVNAM=DEVICE_DESCR    -;Device descriptor
                CHAN=DEVICE_CHANNEL     ;Channel
        CMPW    #SS$_NORMAL,R0          ;Test for normal return
        BSBW    ERROR_CHECK             ;See if all is well
```

```
;
; Now we will write the message to the mailbox using the function
; modifier IO$M_NOW so that we may continue without waiting for a
; read on the mailbox
;

        $QIOW_S FUNC=#IO$_WRITEVBLK!IO$M_NOW,-  ;Write message NOW
                CHAN=MAILBOX_CHANNEL,-  ;To the mailbox channel
                P1=OUT_BOX_BUFFER,-     ;Buffer to write
                P2=#OUT_LENGTH          ;How much to write
        CMPW    #SS$_NORMAL,R0          ;Test for normal return
        BSBW    ERROR_CHECK             ;See if all is well

;
; Now the mailbox is read
;

        $QIOW_S FUNC=#IO$_READVBLK,-    ;read box
                CHAN=MAILBOX_CHANNEL,-  ;Mailbox channel
                IOSB=STATUS,-           ;Define status to receive message length
                P1=IN_BOX_BUFFER,-      ;Where to read it
                P2=#IN_LENGTH           ;How much
        CMPW    #SS$_NORMAL,R0          ;Test for normal return
        BSBW    ERROR_CHECK             ;See if all is well

;
; Now we find out how much mail was in the box and print it to the terminal
; The amount of mail read is held in STATUS+2
;

        MOVZWL  STATUS+2,R2             ;Put byte count into R2
        $QIOW_S FUNC=#IO$_WRITEVBLK,-   ;Function is write
                CHAN=DEVICE_CHANNEL,-   ;To the terminal
                P1=IN_BOX_BUFFER,-      ;Address of buffer to write
                P2=R2,-                 ;How much to write
                P4=#32                  ;Carriage control (1H ,)

;
; we now deassign the channel and exit
;

EXIT:   $DASSGN_S  CHAN=DEVICE_CHANNEL  ;Deassign channel
        RET                             ;Return

;
; This is the error checking part of the program. Normally some kind of
; error recovery would be attempted here but not for this example.
;

ERROR_CHECK:                            ;
        BNEQ    EXIT                    ;Directive failed so exit
        RSB                             ;Else return

        .END    START
```

CHAPTER 8

DMC11 SYNCHRONOUS COMMUNICATIONS LINE INTERFACE DRIVER

This chapter describes the use of the VAX/VMS DMC11 Synchronous Communications Line Interface driver. The DMC11 provides a direct-memory-access (DMA) interface between two computer systems using the DIGITAL Data Communications Message Protocol (see Section 8.1.1 below). The DMC11 supports DMA data transfers of up to 16K bytes at rates of up to 1 million baud for local operation (over coaxial cable) and 56,000 baud for remote operation (using modems). Both full- and half-duplex modes are supported.

The DMC11 is a message-oriented communications line interface that is used primarily to link two separate but cooperating computer systems.

8.1 SUPPORTED DMC11 SYNCHRONOUS LINE INTERFACES

Table 8-1 lists the DMC11 options supported by VAX/VMS.

Table 8-1
Supported DMC11 Options

Type	Use
DMC11-AR with DMC11-FA DMC11-AR with DMC11-DA	Remote DMC11 and EIA or V35/DDS line unit
DMC11-AL with DMC11-MD DMC11-AL with DMC11-MA	Local DMC11 and 1M bps or 56 bps

8.1.1 DIGITAL Data Communications Message Protocol

To ensure reliable data transmission, the DIGITAL Data Communications Message Protocol (DDCMP) has been implemented, using a high-speed microprocessor, on the VAX-11/780 processor. For remote operations, a DMC11 can communicate with a different type of synchronous interface (or even a different type of computer), provided the remote system has implemented DDCMP, version 4.

DDCMP detects errors on the communication line interconnecting the systems using a 16-bit Cyclic Redundancy Check (CRC). Errors are corrected, when necessary, by automatic message retransmission. Sequence numbers in message headers ensure that messages are delivered in the proper order with no omissions or duplications.

The DDCMP specification (Order No. AA-D599A-TC) provides more detailed information on DDCMP.

8.2 DRIVER FEATURES AND CAPABILITIES

DMC11 driver capabilities include:

- A nonprivileged QIO interface to the DMC11. This allows use of the DMC11 as a raw-data channel.
- Unit attention conditions transmitted through attention ASTs and mailbox messages.
- Both full- and half-duplex operation.
- Interface design common to all communications devices supported by VAX/VMS.
- Error logging of all DMC11 microprocessor and line unit errors.
- Online diagnostics.
- Separate transmit and receive quotas.
- Assignment of several read buffers to the device.

The following sections describe mailbox usage and I/O quotas.

8.2.1 Mailbox Usage

The device owner process can associate a mailbox with a DMC11 by using the $ASSIGN system service (see Section 7.1.2). The mailbox is used to receive messages that signal attention conditions about the unit. As illustrated in Figure 8-1, these messages have the following content and format:

- Message type; this can be any one of the following:

Message type	Meaning
MSG$_XM_DATAVL	Data is available
MSG$_XM_SHUTDN	Unit has been shutdown
MSG$_XM_ATTN	A disconnect, timeout, or data check occurred

  The $MSGDEF macro is used to define message types

- Physical unit number of the DMC11
- Size (count) of the ASCII device name string
- Device name string

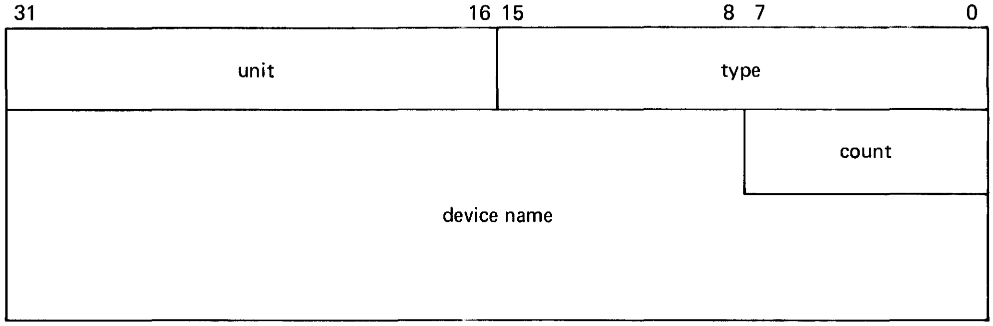


Figure 8-1 Mailbox Message Format

8.2.2 Quotas

Transmit operations are considered direct I/O operations and are limited by the process's direct I/O quota.

The quotas for the receive buffer free list (see Section 8.4.3.4) are the process's buffered I/O count and buffered I/O byte limit. After start up, the transient byte count and the buffered I/O byte limit are adjusted.

8.2.3 Power Failure

When a system power failure occurs, no DMC11 recovery is possible. The device is in a fatal error state and is shut down.

8.3 DEVICE INFORMATION

Users can obtain information on device characteristics by using the $GETCHN and $GETDEV system services (see Section 1.10). The information is returned in a user-specified buffer shown in Figure 8-2. Only the first three longwords of the buffer are shown in Figure 8-2 (Figure 1-9 shows the entire buffer).

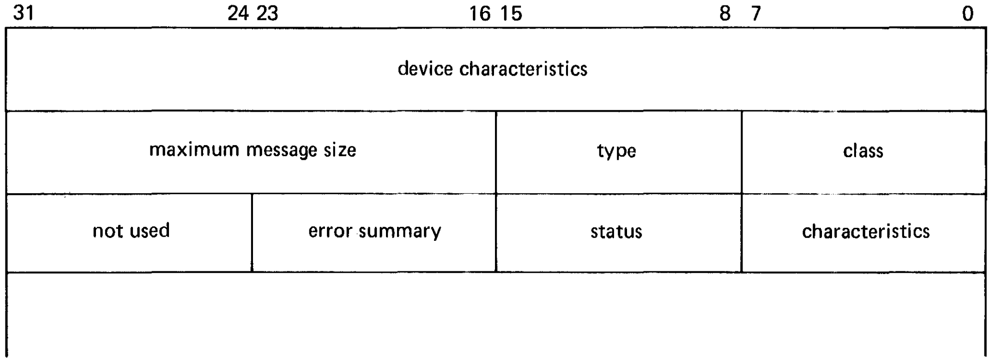


Figure 8-2 DMC11 Information

The first longword in the buffer contains the device characteristics values listed in Table 8-2. The $DEVDEF macro defines these values.

Table 8-2
DMC11 Device Characteristics

Dynamic bit (Conditionally Set)	Meaning
DEV$M_NET	Network device
Static bits (Always Set)	
DEV$M_ODV	Output device
DEV$M_IDV	Input device

The second longword contains information on the device class and type, and the maximum message size. The device class for the DMC11 is DC$_SCOM. Table 8-3 lists the device types. The device class and types are defined by the $DCDEF macro.

Table 8-3
DMC11 Device Types

Device Type	Meaning[1]
DT$_XM_ARDA	DMC11-AR with DMC11-DA
DT$_XM_ARFA	DMC11-AR with DMC11-FA
DT$_XM_ALMD	DMC11-AL with DMC11-MD
DT$_XM_ALMA	DMC11-AL with DMC11-MA

[1] Table 8-1 describes the different device types

The maximum message size is the maximum send or receive message size for the unit. Messages greater than 512 bytes on modem controlled lines are more prone to transmission errors and therefore may require more retransmissions.

The third longword contains unit characteristics and status, and an error summary.

Unit characteristics bits govern the DDCMP operating mode. They are defined by the $XMDEF macro and can be read or set. Table 8-4 lists the unit characteristics values and their meanings.

Table 8-4
DMC11 Unit Characteristics

Characteristic	Meaning[1]
XM$M_CHR_MOP	DDCMP maintenance mode
XM$M_CHR_SLAVE	DDCMP half-duplex slave station
XM$M_CHR_HDPLX	DDCMP half-duplex
XM$M_CHR_LOOPB	DDCMP loop back
XM$M_CHR_MBX	Shows the status of the mailbox that can be associated with the unit; if this bit is set, the mailbox is enabled to receive messages signaling unsolicited data. (This bit can also be changed as a subfunction of read or write QIO functions)

[1] Section 8.1.1 describes DDCMP

The status bits show the status of the unit and the line. The values are defined by the $XMDEF macro. They can be read, set, or cleared as indicated. Table 8-5 lists the status values and their meanings.

Table 8-5
DMC11 Unit and Line Status

Status	Meaning
XM$M_STS_ACTIVE	Protocol is active. This bit is set when IO$_SETMODE!IO$_STARTUP is done and cleared when the unit is shut down. (Read only.)
XM$M_STS_TIMO	Timeout. If set, indicates that the receiving computer is unresponsive. DDCMP time outs. (Read or clear.)
XM$M_STS_ORUN	Data overrun. If set, indicates that a message was received but lost due to the lack of a receive buffer. (Read or clear.)
XM$M_STS_DCHK	Data check. If set, indicates that a retransmission threshold has been exceeded. (Read or clear.)
XM$M_STS_DISC	If set, indicates that the Data Set Ready (DSR) modem line went from on to off. (Read or clear.)

The error summary bits are set only when the driver must shut down the DMC11 because a fatal error occurred. These are read-only bits that are cleared by any of the IO$_SETMODE functions (see Section 8.4.3). The XM$M_STS_ACTIVE status bit is clear if any error summary bit is set. Table 8-6 lists the error summary bit values and their meanings.

Table 8-6
Error Summary Bits

Error Summary Bit	Meaning
XM$M_ERR_MAINT	DDCMP maintenance message received
XM$M_ERR_START	DDCMP START message received
XM$M_ERR_LOST	Data was lost when a message was received that was longer than the specified maximum message size.
XM$M_ERR_FATAL	An unexpected hardware/software error occurred.

8.4 DMC11 FUNCTION CODES

The basic DMC11 function codes are read, write, and set mode. All three functions take function modifiers.

8.4.1 Read

VAX/VMS provides three read function codes:

- IO$_READLBLK - read logical block
- IO$_READPBLK - read physical block
- IO$_READVBLK - read virtual block

Received messages are multi-buffered in system dynamic memory and then copied to the user's address space when the read operation is performed.

The QIO arguments for the three function codes are:

- P1 -- the starting virtual address of the buffer that is to receive data
- P2 -- the size of the receive buffer in bytes

The read QIO functions can take two function modifiers:

- IO$M_DSABLMBX - disable use of the associated mailbox for unsolicited data notification
- IO$M_NOW - complete the read operation immediately if no message is available

8.4.2 Write

VAX/VMS provides three write QIO function codes:

- IO$_WRITELBLK - write logical block
- IO$_WRITEPBLK - write physical block
- IO$_WRITEVBLK - write virtual block

Transmitted messages are sent directly from the requesting process's buffer.

The QIO arguments for the three function codes are:

- P1 -- the starting virtual address of the buffer containing the data to be transmitted
- P2 -- the size of the buffer in bytes

The message size specified by P2 cannot be larger than the maximum send message size for the unit (see Section 8.3). If a message larger than the maximum size is sent, a status of SS$_DATAOVERUN is returned in the I/O status block.

The write QIO functions can take one function modifier:

- IO$M_ENABLMBX - enable use of the associated mailbox

8.4.3 Set Mode

Set mode operations are used to perform protocol, operational, and program/driver interface operations with the DMC11. VAX/VMS defines five types of set mode functions:

- Set Mode
- Set Characteristics
- Enable Attention AST
- Set Mode and Shut Down Unit
- Set Mode and Start Unit

8.4.3.1 **Set Mode and Set Characteristics** - These functions set device characteristics such as maximum message size. VAX/VMS provides two function codes:

- IO$_SETMODE - set mode (requires logical I/O privilege)
- IO$_SETCHAR - set characteristics (requires physical I/O privilege)

One argument is used with these function codes:

P1 -- the virtual address of the quadword characteristics buffer block if the characteristics are to be set. If this argument is zero, only the unit status and characteristics are returned in the I/O status block (see Section 8.5). Figure 8-3 shows the P1 characteristics block.

31 24	23 16	15 8	7 0
maximum message size		type	class
not used	error summary	status	characteristics

Figure 8-3 P1 Characteristics Block

In the buffer designated by P1 the device class is DC$_SCOM. Table 8-3 (in Section 8.3) lists the device types. The maximum message size describes the maximum send or receive message size.

The second longword contains device/function-dependent characteriscs: unit characteristics, status, and error summary bits. Any of the characteristics values and some of the status values can be set or cleared (see Tables 8-4, 8-5, and 8-6).

If the unit is active (XM$M_STS_ACTIVE is set), the action of a Set Mode or Set Characteristics function with a characteristics buffer is to clear the status bits or the error summary bits. If the unit is not active, the status bits or the error summary bits can be cleared, and the maximum message size, type, device class, and unit characteristics can be changed.

8.4.3.2 **Enable Attention AST** - This function enables an AST to be queued when an attention condition occurs on the unit. An AST is queued when the driver sets or clears either an error summary bit or any of the unit status bits, or when a message is available and there is no waiting read request. The Enable Attention AST function is legal at any time, regardless of the condition of the unit status bits.

VAX/VMS provides two function codes:

- IO$_SETMODE!IO$M_ATTNAST - enable attention AST

- IO$_SETCHAR!IO$M_ATTNAST - enable attention AST

Enable Attention AST is a single (one-time) enable. After the AST occurs, it must be explicitly reenabled by the function before the AST can occur again. The function code is also used to disable the AST. The function is subject to AST quotas.

The Enable Attention AST functions take the following device/function dependent arguments:

- P1 -- address of AST service routine or 0 for disable

- P2 -- (ignored)

- P3 -- access mode to deliver AST

The AST service routine is called with an argument list. The first argument is the current value of the device/function dependent characteristics longword shown in Figure 8-3. The access mode specified by P3 is maximized with the requester's access mode.

8.4.3.3 **Set Mode and Shut Down Unit** - This function stops the operation on an active unit (XM$M_STS_ACTIVE must be set) and then resets the unit characteristics.

VAX/VMS provides two function codes:

- IO$_SETMODE!IO$M_SHUTDOWN - shut down unit
- IO$_SETCHAR!IO$M_SHUTDOWN - shut down unit

These codes take one device/function dependent argument:

> P1 -- the virtual address of the quadword characteristics block (Figure 8-3) if modes are to be set after shutdown. P1 is 0 if modes are not to be set after shutdown.

These functions stop the DMC11 microprocessor and release all outstanding message blocks; any messages that have not been read are lost. The characteristics are reset after shutdown. Except for the signaling of attention ASTs and mailbox messages, the action of these functions is the same as the action of the driver when shutdown occurs because of a fatal error.

8.4.3.4 **Set Mode and Start Unit** - This function sets the characteristics and starts the protocol on the associated unit. VAX/VMS provides two function codes:

- IO$_SETMODE!IO$M_STARTUP - start unit
- IO$_SETCHAR!IO$M_STARTUP - start unit

These codes take the following device/function dependent arguments:

- P1 -- the virtual address of the quadword characteristics block (Figure 8-3) if the characteristics are to be set. Characteristics are set before the device is started.
- P2 -- (ignored).
- P3 -- the number of pre-allocated receive-message blocks to ensure the availability of buffers to receive messages.

The total quota taken from the process's buffered I/O byte count quota is the DMC11 work space plus the number of receive-message buffers specified by P3 times the maximum message size. For example, if six 200-byte, buffers are required, the total quota taken is 1456 bytes:

```
   256  (DMC11 work space)
+ 1200  (number of buffers X buffer size)
  ----
  1456  (total quota taken)
```

This quota is returned to the process when shutdown occurs.

Receive-message blocks are used by the driver to receive messages that arrive independent of QIO read request timing. When a message arrives, it is matched with any outstanding read requests. If there are no outstanding read requests, the message is queued and an attention AST or mailbox message is generated. (IO$_SETMODE!IO$M_ATTNAST or IO$_SETCHAR!IO$M_ATTNAST must be set to enable an attention AST; IO$M_ENABLMBX must be used to enable a mailbox message.)

When read, the receive-message block is returned to the receive-message "free list" defined by P3. If the "free list" is empty, no receive-messages are possible. In this case, a data lost condition can be generated if a message arrives. This nonfatal condition is reported by device-dependent data and an attention AST.

8.5 I/O STATUS BLOCK

The I/O status block (IOSB) usage for all DMC11 QIO functions is shown in Figure 8-4. Table 8-7 lists the status returns for these functions.

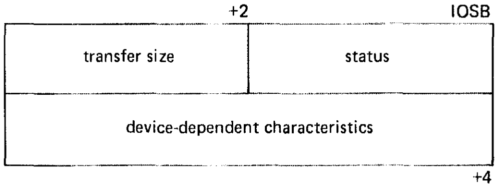


Figure 8-4 IOSB Content

In Figure 8-4, the transfer size at IOSB+2 is the actual number of bytes transferred. Table 8-4 lists the device-dependent characteristics returned at IOSB+4. These characteristics can also be obtained by using the $GETCHN and $GETDEV system services (see Section 8.3).

Table 8-7
Status Returns for DMC11

Status	Meaning
SS$_ABORT	Fatal hardware error or I/O canceled in progress.
SS$_DATAOVERUN	Message received overran buffer allocated (read), or message too big (write).
SS$_ENDOFFILE	No data available (read) when IO$M_NOW was specified.
SS$_NORMAL	Operation was successfully completed (read, write, or set modes).
SS$_DEVOFFLINE	Device protocol not started (read or write). The function is inconsistent with the current state of the unit (Set Mode).
SS$_DEVACTIVE	The function is inconsistent with the current state of the unit.

CHAPTER 9

QIO INTERFACE TO FILE SYSTEM ACPS

An ancillary control process (ACP) is a process that interfaces between the user process and the driver, and performs functions that supplement the driver's functions. Virtual I/O operations involving file-structured devices (disks and magnetic tapes) often require ACP intervention. In most cases, ACP intervention is requested by VAX-11 Record Management Services (RMS) and is transparent to the user process. However, user processes can request ACP functions directly by issuing a QIO request and specifying an ACP function code, as shown in Figure 9-1.

The DECnet User's Guide describes network ACP (NETACP) interface operations.

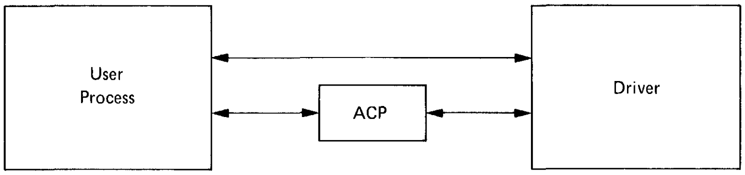


Figure 9-1 ACP QIO Interface

This chapter describes the QIO interface to ACPs for disk and magnetic tape devices (file system ACPs). The sample program in Chapter 4 performs QIO operations to the magnetic tape ACP.

9.1 FILE INFORMATION BLOCK

The File Information Block (FIB) contains much of the information that is exchanged between the user process and the ACP. Figure 9-2 shows the format of the FIB. Because the FIB is passed by a descriptor (Pl in Figure 9-7), its length can vary. Thus a short FIB can be used in ACP calls that do not need arguments toward the end of the FIB. The ACP automatically zero-extends a short FIB. Figure 9-3 shows the format of a typical short FIB, in this case one that would be used to open an existing file. Table 9-1 lists the values of these FIB fields.

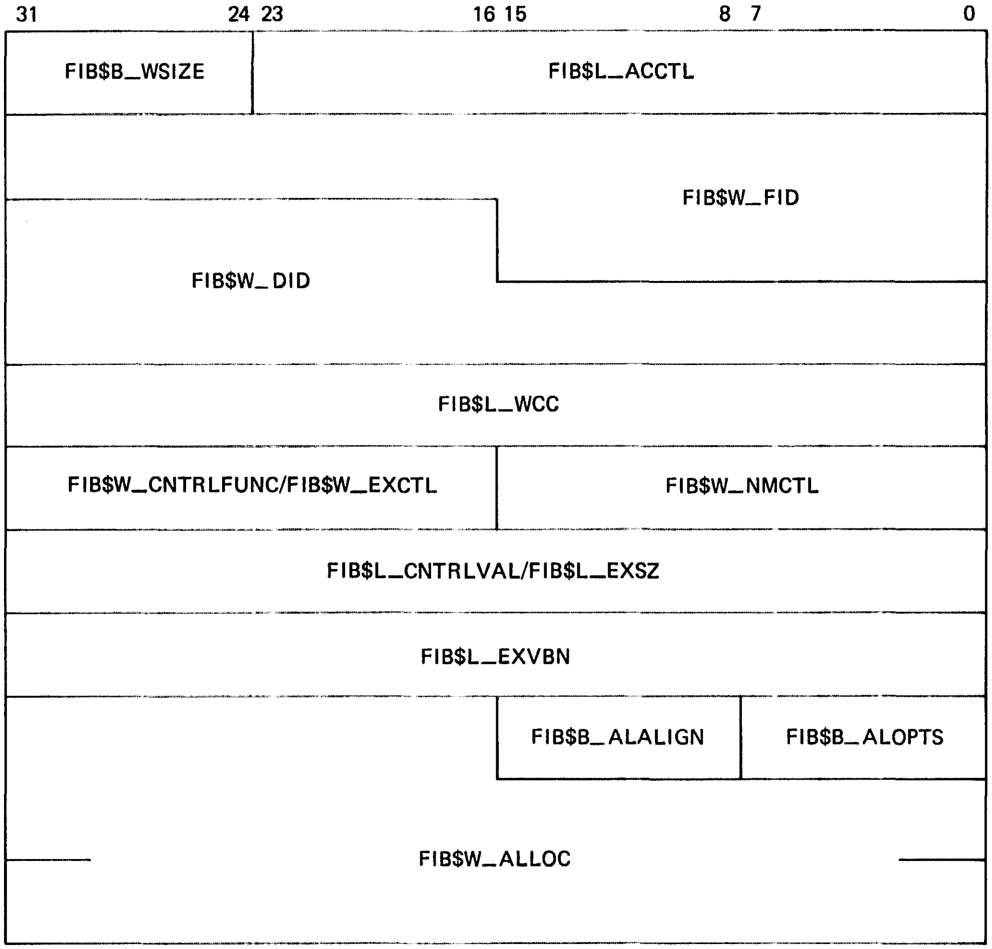


Figure 9-2 File Information Block Format

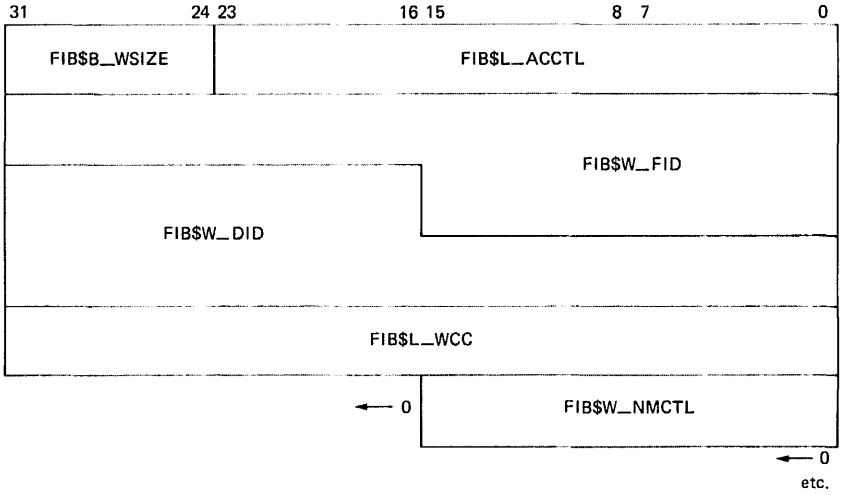


Figure 9-3 Typical Short File Information Block

Table 9-1
Contents of the File Information Block

Field	Field Values	Meaning
FIB$L_ACCTL		Specifies field values that control access to the file. The following bits are defined:
	FIB$M_WRITE	Set for write access; clear for read-only access.
	FIB$M_NOREAD	Set to deny read access to others. (The user also must have write privilege to the file.)
	FIB$M_NOWRITE	Set to deny write access to others.
	FIB$M_NOTRUNC	Set to prevent the file from being truncated; clear to allow truncation.
	FIB$M_DLOCK	Set to enable deaccess lock (close check). Only for disk devices. Used to flag a file as inconsistent in the event the program currently modifying the file terminates abnormally. If the program then closes the file without performing a write attributes operation, the file is marked as locked and cannot be accessed until it is unlocked.
	FIB$M_SEQONLY	Set for sequential-only access. Only for disk devices.
	FIB$M_REWIND	Set to rewind magnetic tape before access.
	FIB$M_CURPOS	Set to create magnetic tape file at current position (note: a magnetic tape file will be created at the end of the volume set if neither FIB$M_REWIND nor FIB$M_CURPOS is set). If the tape is not positioned at the end of a file, FIB$M_CURPOS creates a file at the next file position.
	FIB$M_UPDATE	Set to position at start of a magnetic tape file when opening file for write; clear to position at end-of-file.

(continued on next page)

Table 9-1 (Cont.)
Contents of the File Information Block

Field	Field Values	Meaning
FIB$L_ACCTL (Cont.)	FIB$M_PRSRV_ATR	Set to use span attribute recorded in system-dependent attribute area. Only for magnetic tape devices.
	FIB$M_READCK	Set to enable read checking of the file.
	FIB$M_WRITECK	Set to enable write checking of the file.
	FIB$M_EXECUTE	Set to access the file in execute mode. The protection check is made against the EXECUTE bit instead of the READ bit. Valid only for requests issued from EXEC or KERNEL mode.
	FIB$M_RMSLOCK	Set to declare RMS record locking on the file. All users of a file must employ the same configuration of this bit, that is, if a file is opened with RMS record locking, other non-RMS users are locked out. Valid only for requests issued from EXEC or KERNEL mode.
FIB$B_WSIZE		Controls the size of the file window used to map a disk file. The ACP will use the volume default if FIB$B_WSIZE is 0. A value of 1 to 127 indicates the number of retrieval pointers to be allocated to the window. A value of -1 indicates that the window should be as large as necessary to map the entire file.
FIB$W_FID		Specifies the file identification. The user supplies the file identifier when it is known; the ACP returns the file identifier when it becomes known, for example, as a result of a create or directory lookup. The following subfields are defined:
	FIB$W_FID_NUM	File number

(continued on next page)

Table 9-1 (Cont.)
Contents of the File Information Block

Field	Field Values	Meaning
FIB$W_FID (Cont.)	FIB$W_FID_SEQ	File sequence number
	FIB$W_FID_RVN	Relative volume number
FIB$W_DID		Contains the file identifier of the directory file. The following subfields are defined:
	FIB$W_DID_NUM	File number
	FIB$W_DID_SIQ	File sequence number
	FIB$W_DID_RVN	Relative volume number
FIB$L_WCC		Maintains position context when processing wild card directory operations
FIB$W_NMCTL		Controls the processing of a name string in a directory operation. The following bits are defined:
	FIB$M_WILD	Set if name string contains wild cards
	FIB$M_ALLNAM	Set to match all name field values
	FIB$M_ALLTYP	Set to match all field type values
	FIB$M_ALLVER	Set to match all version field values
	FIB$M_NEWVER	Set to create file of same name with next higher version number. Only for disk devices.
	FIB$M_SUPERSEDE	Set to supersede an existing file of the same name type, and version. Only for disk devices.
	FIB$M_FINDFID	Set to search a directory for the file identifier in FIB$W_FID
	FIB$M_LOWVER	Set on return from a CREATE if a lower numbered version of the file exists. Only for disk devices.

(continued on next page)

Table 9-1 (Cont.)
Contents of the File Information Block

Field	Field Values	Meaning
FIB$W_NMCTL (Cont.)	FIB$M_HIGHVER	Set on return from a CREATE if if a higher numbered version of the file exists. Only for disk devices.
FIB$W_EXCTL		Specifies extend control for disk devices. The following bits are defined:
	FIB$M_EXTEND [1]	Set to enable extension
	FIB$M_TRUNC [1]	Set to enable truncation
	FIB$M_NOHDREXT	Set to inhibit generation of extension file headers
	FIB$M_ALCON	Allocate contiguous space
	FIB$M_ALCONB	Allocate contiguous space; best effort
	FIB$M_FILCON	Mark file contiguous
	FIB$M_ALDEF	Allocate the extend size (FIB$L_EXSZ) or the system default, whichever is greater
	FIB$M_MARKBAD	Set to deallocate blocks to the bad block file during a truncate operation
	FIB$M_ALLOCATR	Set if placement control data is present in the attribute list. For compatibility mode use.
FIB$W_CNTRLFUNC		Controls magnetic tape functions and disk quota file operations. This field overlays FIB$W_EXCTL. In an ACPCONTROL function, the FIB$W_CNTRLFUNC field can contain one of the following values:
	FIB$C_REWINDFIL	Rewind to beginning of file
	FIB$C_POSEND	Position to end of volume set
	FIB$C_NEXTVOL	Force next volume

[1] Only one of these can be set at one time; that is, extension cannot be enabled at the same time truncation is enabled, and vice versa.

(continued on next page)

Table 9-1 (Cont.)
Contents of the File Information Block

Field	Field Values	Meaning
FIB$W_CNTRLFUNC (Cont.)	FIB$C_SPACE	Space n blocks forward or in reverse
	FIB$C_REWINDVOL	Rewind to beginning of volume set
	FIB$C_ENA_QUOTA	Enable the disk quota file[1]
	FIB$C_DSA_QUOTA	Disable the disk quota file[1]
	FIB$C_ADD_QUOTA	Add an entry to the disk quota file[1]
	FIB$C_EXA_QUOTA	Examine a disk quota file entry[1]
	FIB$C_MOD_QUOTA	Modify a disk quota file entry[1]
	FIB$C_REM_QUOTA	Remove a disk quota file entry[1]
	FIB$C_LOCK_VOL	Allocation lock the volume[1]
	FIB$C_UNLK_VOL	Unlock the volume. Cancels FIB$C_LOCK_VOL.[1]
FIB$L_EXSZ		Specifies the number of blocks to allocate to, or remove from, a disk file depending on the FIB$W_EXCTL field configuration. For truncate operations, this field must contain 0. The number of blocks actually allocated or removed is returned in this longword. The value may differ from the user-requested value because of adjustments for cluster boundaries. More blocks are allocated and fewer blocks removed to meet cluster boundaries.

[1] Table 9-6 describes the disk quota and lock/unlock bits in greater detail.

(continued on next page)

Table 9-1 (Cont.)
Contents of the File Information Block

Field	Field Values	Meaning
FIB$L_CNTRLVAL		Specifies magnetic tape block movements or disk quota file functions. This field overlays FIB$L_EXSZ. If FIB$C_SPACE is indicated, the FIB$L_CNTRLVAL field specifies the number of magnetic tape blocks to space forward if positive or space backward if negative. The following bits are defined for disk quota file operations:
	FIB$M_ALL_MEM	Wild card through the disk quota file and match all UIC members
	FIB$M_ALL_GRP	Wild card through the disk quota file and match all UIC groups
	FIB$M_MOD_PERM	If FIB$C_MOD_QUOTA is specified, change the permanent disk quota
	FIB$M_MOD_USE	If FIB$C_MOD_QUOTA is specified, change the usage data. The volume must be locked by FIB$C_LOCK_VOL. This operation requires write access to the disk quota file.
FIB$L_EXVBN		Specifies the starting disk file virtual block number at which the allocated blocks are to appear in an extend operation, or the first virtual block number to be removed in a truncate operation. For extend operations, this field must contain either the end-of-file block number plus 1, or 0. For truncate operations, this field specifies the first virtual block number to be removed. The actual starting virtual block number of the extend or truncate operation is returned in this field.

(continued on next page)

Table 9-1 (Cont.)
Contents of the File Information Block

Field	Field Values	Meaning
FIB$B_ALOPTS		Contains option bits that control the placement of allocated blocks. The following bits are defined:
	FIB$M_EXACT	Set to require exact placement; clear to specify approximate placement.
	FIB$M_ONCYL	Set to locate allocated space within a cylinder
FIB$B_ALALIGN		Contains the interpretation mode of the allocation (FIB$W_ALLOC) field. One of the following values can be specified:
	(zero)	No placement data. The remainder of the allocation field is ignored.
	FIB$C_CYL	Location is specified as a dummy longword, followed by a word Relative Volume Number (RVN), followed by a longword cylinder number.
	FIB$C_LBN	Location is specified as a dummy longword, followed by a word RVN, followed by a longword Logical Block Number (LBN).
	FIB$C_VBN	Location is specified as three dummy words followed by a longword Virtual Block Number (VBN) of the file being extended. A zero VBN or one that fails to map indicates the end of the file.
	FIB$C_RFI	Location is specified as a 3-word file ID, followed by a longword VBN in that file. A zero file ID indicates the file being extended. A zero VBN or one that fails to map indicates the end of that file.

(continued on next page)

Table 9-1 (Cont.)
Contents of the File Information Block

Field	Field Values	Meaning
FIB$W_ALLOC		Contains the desired physical location of the blocks being allocated. Interpretation of the field is controlled by the FIB$B_ALALIGN field. The following subfields are defined:
	FIB$W_LOC_FID	3-word related file ID for RFI placement
	FIB$W_LOC_NUM	Related file number
	FIB$W_LOC_SEQ	Related file sequence number
	FIB$W_LOC_RVN	Related file RVN or placement RVN
	FIB$W_LOC_ADDR	Placement LBN, cylinder, or VBN

Table 9-2 shows which FIB fields and field values are used in the respective QIO functions. Some of the FIB fields and values are applicable only to disk devices or only to magnetic tape devices. See Table 9-1.

Table 9-2
FIB Argument Usage in ACP QIO Functions

I/O Function	Applicable Arguments	
	FIB Field	Field Values
IO$_CREATE	FIB$L_ACCTL	FIB$M_WRITE FIB$M_NOREAD FIB$M_NOWRITE FIB$M_NOTRUNC FIB$M_DLOCK FIB$M_SEQONLY FIB$M_REWIND FIB$M_CURPOS FIB$M_UPDATE FIB$M_PRSRV_ATR FIB$M_READCK FIB$M_WRITECK FIB$M_EXECUTE FIB$M_RMSLOCK

(continued on next page)

Table 9-2 (Cont.)
FIB Argument Usage in ACP QIO Functions

I/O Function	Applicable Arguments	
	FIB Field	Field Values
IO$_CREATE (CONT.)	FIB$B_WSIZE	
	FIB$W_FID [1]	FIB$W_FID_NUM FIB$W_FID_SEQ FIB$W_FID_RVN
	FIB$W_DID	FIB$W_DID_NUM FIB$W_DID_SEQ FIB$W_DID_RVN
	FIB$W_NMCTL	FIB$M_NEWVER FIB$M_SUPERSEDE FIB$M_FINDFID FIB$M_LOWVER [2] FIB$M_HIGHVER [2]
	FIB$W_EXCTL	FIB$M_EXTEND FIB$M_NOHDREXT FIB$M_ALCON FIB$M_ALCONB FIB$M_FILCON FIB$M_ALDEF FIB$M_ALLOCATR
	FIB$L_EXSZ	
	FIB$B_ALOPTS	FIB$M_EXACT FIB$M_ONCYL
	FIB$B_ALALIGN	(zero) FIB$C_CYL FIB$C_LBN FIB$C_VBN FIB$C_RFI
	FIB$W_ALLOC	FIB$W_LOC_FID FIB$W_LOC_NUM FIB$W_LOC_SEQ FIB$W_LOC_RVN FIB$W_LOC_ADDR

[1] If FIB$W_DID = 0 and IO$M_CREATE is not set; FIB$W_FID is an output argument if IO$M_CREATE is set.

[2] Output argument

(continued on next page)

Table 9-2 (Cont.)
FIB Argument Usage in ACP QIO Functions

I/O Function	Applicable Arguments	
	FIB Field	Field Values
IO$_ACCESS	FIB$L_ACCTL	FIB$M_WRITE FIB$M_NOREAD FIB$M_NOWRITE FIB$M_NOTRUNC FIB$M_DLOCK FIB$M_SEQONLY FIB$M_REWIND FIB$M_CURPOS FIB$M_UPDATE FIB$M_PRSRV_ATR FIB$M_READCK FIB$M_WRITECK FIB$M_EXECUTE FIB$M_RMSLOCK
	FIB$B_WSIZE	
	FIB$W_FID [1]	FIB$W_FID_NUM FIB$W_FID_SEQ FIB$W_FID_RVN
	FIB$W_DID	FIB$W_DID_NUM FIB$W_DID_SEQ FIB$W_DID_RVN
	FIB$L_WCC [2]	
	FIB$W_NMCTL	FIB$M_WILD FIB$M_ALLNAM FIB$M_ALLTYP FIB$M_ALLVER FIB$M_FINDFID
IO$_DEACCESS	FIB$W_EXCTL	FIB$M_TRUNC
	FIB$L_EXVBN	
IO$_MODIFY	FIB$W_FID [1]	FIB$W_FID_NUM FIB$W_FID_SEQ FIB$W_FID_RVN
	FIB$W_DID	FIB$W_DID_NUM FIB$W_DID_SEQ FIB$W_DID_RVN

1. If FIB$W_DID is 0; FIB$W_FID is an output argument if FIB$W_DID is not 0.

2. If FIB$M_WILD is set.

(continued on next page)

Table 9-2 (Cont.)
FIB Argument Usage in ACP QIO Functions

I/O Function	Applicable Arguments	
	FIB Field	Field Values
IO$_MODIFY (Cont.)	FIB$L_WCC[1]	
	FIB$W_NMCTL	FIB$M_WILD FIB$M_ALLNAM FIB$M_ALLTYP FIB$M_ALLVER FIB$M_FINDFID
	FIB$W_EXCTL	FIB$M_EXTEND[3] FIB$M_TRUNC FIB$M_NOHDREXT FIB$M_ALCON FIB$M_ALCONB FIB$M_FILCON FIB$M_ALDEF FIB$M_MARKBAD FIB$M_ALLOCATR
	FIB$L_EXSZ	
	FIB$L_EXVBN	
	FIB$B_ALOPTS	FIB$M_EXACT FIB$M_ONCYL
	FIB$B_ALALIGN	(zero) FIB$C_CYL FIB$C_LBN FIB$C_VBN FIB$C_RFI
	FIB$W_ALLOC	FIB$W_LOC_FID FIB$W_LOC_NUM FIB$W_LOC_SEQ FIB$W_LOC_RVN FIB$W_LOC_ADDR
IO$_DELETE	FIB$W_FID[2]	FIB$W_FID_NUM FIB$W_FID_SEQ FIB$W_FID_RVN
	FIB$W_DID	FIB$W_DID_NUM FIB$W_DID_SEQ FIB$W_DID_RVN

[1] If FIB$M_WILD is set.

[2] If FIB$W_DID is 0; FIB$W_DID is an output argument if FIB$W_DID is not 0.

[3] Only FIB$M_EXTEND or FIB$_TRUNC can be set at any given time; they cannot both be set at the same time.

(continued on next page)

Table 9-2 (Cont.)
FIB Argument Usage in ACP QIO Functions

I/O Function	Applicable Arguments	
	FIB Field	Field Values
IO$_DELETE (Cont.)	FIB$L_WCC [1]	
	FIB$W_NMCTL	FIB$M_WILD FIB$M_ALLNAM FIB$M_ALLTYP FIB$M_ALLVER FIB$M_FINDFID
IO$_MOUNT (no arguments used)		
IO$_ACPCONTROL	FIB$W_CNTRLFUNC	FIB$C_REWINDFIL FIB$C_POSEND FIB$C_NEXTVOL FIB$C_SPACE FIB$C_REWINDVOL FIB$C_ENA_QUOTA FIB$C_DSA_QUOTA FIB$C_ADD_QUOTA FIB$C_EXA_QUOTA FIB$C_MOD_QUOTA FIB$C_REM_QUOTA FIB$C_LOCK_VOL FIB$C_UNLK_VOL
	FIB$L_CNTRLVAL	FIB$M_ALL_MEM FIB$M_ALL_GRP FIB$M_MOD_PERM FIB$M_MOD_USE

1. If FIB$M_WILD is set.

9.2 ATTRIBUTE CONTROL BLOCK

The attribute control block contains the codes that control the reading and writing of file attributes, for example, file protection and record attributes. Device/function-dependent argument P5 specifies the address of this list. The list consists of a variable number of 2-longword control blocks, terminated by a zero longword, as shown in Figure 9-4. The maximum number of attribute control blocks in one list is 14. Table 9-3 describes the attribute control block fields.

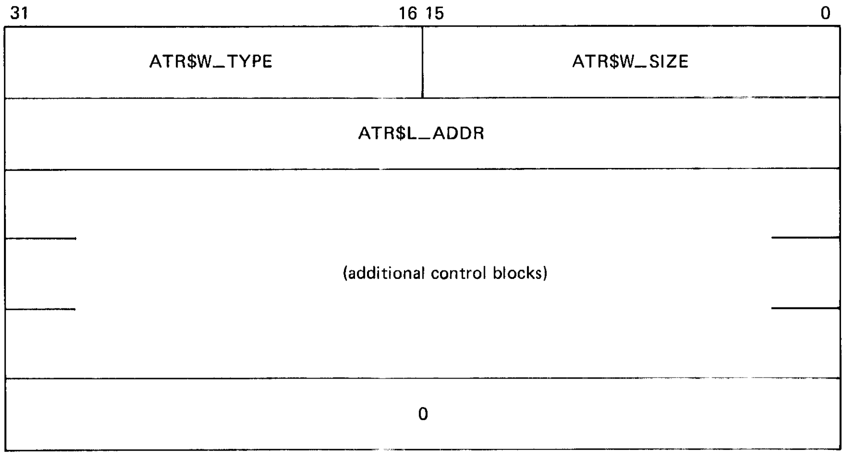


Figure 9-4 Attribute Control Block Format

Table 9-3
Attribute Control Block Fields

Field	Meaning
ATR$W_SIZE	Specifies the number of bytes of the attribute to be transferred. Legal values are from 0 to the maximum size of the particular attribute (see Table 9-4).
ATR$W_TYPE	Identifies the individual attribute to be read or written.
ATR$W_ADDR	Contains the buffer address of the user's memory space to or from which the attribute is to be transferred. The particular I/O function determines whether the attribute is read or written, as follows: I/O Function Operation Create Write Access Read Deaccess Write Modify Write Delete Not used Mount Not used ACP Control Not used

Table 9-4 lists the valid attributes for ACP QIO functions. The maximum size (in bytes) is determined by the required attribute configuration. For example, the file name uses only 6 bytes, but is always accompanied by the file type and file version - so a total of 10 bytes is required. Each attribute has two names: one for the code (for example, ATR$C_UCHAR) and one for the size (for example, ATR$S_UCHAR).

Table 9-4
ACP QIO Attributes

Attribute Name	Maximum Size (bytes)	Meaning
(unnamed)[2]	5	Two-byte file owner UIC plus the next attribute and the first byte of ATR$C_UCHAR. Used for compatibility mode only.
(unnamed)[2]	3	Two-byte file protection plus the first byte of ATR$C_UCHAR. Used for compatibility mode only.
ATR$C_UCHAR[3] ATR$S_UCHAR	4	Four-byte file characteristics.
ATR$C_RECATTR[3] ATR$S_RECATTR	32	Record attribute area. Section 9.2.1 describes the record attribute area in detail.
ATR$C_FILNAM ATR$S_FILNAM	10	Six-byte Radix-50 file name plus ATR$C_FILTYP and ATR$C_FILVER.
ATR$C_FILTYP ATR$S_FILTYP	4	Two-byte Radix-50 file type plus ATR$C_FILVER
ATR$C_FILVER ATR$S_FILVER	2	Two-byte binary version number.
ATR$C_EXPDAT[2] ATR$S_EXPDAT	7	Expiration date in ASCII.
ATR$C_STATBLK[1] ATR$S_STATBLK	10	Statistics block. Section 9.2.2 describes the statistics block in detail.
ATR$C_HEADER[1] ATR$S_HEADER	512	Complete file header.
ATR$C_BLOCKSIZE ATR$S_BLOCKSIZE	2	Magnetic tape block size.
ATR$C_ASCDATES[2] ATR$S_ASCDATES	35	Revision count (2 binery bytes), revision date, creation date, and expiration date, in ASCII. Format = DDMMMYY (revision date), HHMMSS (time), DDMMMYY (creation date), HHMMSS (time), DDMMMYY (expiration date), HHMMSS (time). The format contains no embedded spaces or commas: DDMMMYYHHMMSSDDMMMYYHHMMSSDDMMMYYHHMMSS

[1] Read-only

[2] Protected (can be written to only by system or owner)

[3] Locked (can not be written to while the file is locked)

(continued on next page)

Table 9-4 (Cont.)
ACP QIO Attributes

Attribute Name	Maximum Size (bytes)	Meaning
ATR$C_ALCONTROL ATR$S_ALCONTROL	14	
ATR$C_ASCNAME ATR$S_ASCNAME	20	File name, type, and version, in ASCII, including punctuation: name.typ;version
ATR$S_CREDATE[2] ATR$S_CREDATE	8	64-bit creation date and time.
ATR$C_REVDATE[2] ATR$S_REVDATE	8	64-bit revision date and time.
ATR$C_EXPDATE[2] ATR$S_EXPDATE	8	64-bit expiration date and time.
ATR$C_UIC[2] ATR$S_UIC	4	4-byte file owner UIC.
ATR$C_FPRO[2] ATR$S_FPRO	2	File protection.
ATR$C_ACLEVEL[2] ATR$S_ACLEVEL	1	File access level.
ATR$C_UIC_RO[1]	4	4-byte file owner UIC

1. Read-only

2. Protected (can be written to only by system or owner)

9.2.1 ACP QIO Record Attributes Area

Figure 9-5 shows the format of the record attributes area.

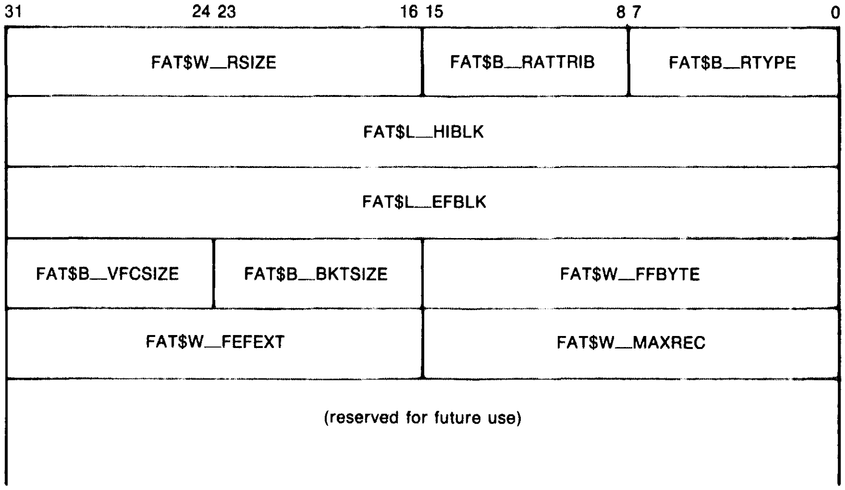


Figure 9-5 ACP QIO Record Attributes Area

Table 9-5 lists the record attributes values and their meanings.

Table 9-5
ACP Record Attributes Values

Field Value	Meaning
FAT$B_RTYPE	Record type. The following bit values are defined: FAT$C_FIXED Fixed record type FAT$C_VARIABLE Variable length FAT$C_VFC Variable and fixed control
FAT$B_RATTRIB	Record attributes. The following bit values are defined: FAT$M_FORTRANCC FORTRAN carriage control FAT$M_IMPLIEDCC Implied carriage control FAT$M_PRINTCC Print file carriage control FAT$M_NOSPAN No spanned records
FAT$W_RSIZE	Record size in bytes
FAT$L_HIBLK [1]	Highest allocated VBN
FAT$L_EFBLK [1]	End-of-file VBN
FAT$W_FFBYTE	First free byte in FAT$L_EFBLK
FAT$B_BKTSIZE	Bucket size in blocks
FAT$B_VFCSIZE	Size in bytes of fixed length control for VFC records
FAT$W_MAXREC	Maximum record size in bytes
FAT$W_DEFEXT	Default extend quantity

[1] Inverted format field

9.2.2 ACP QIO Attributes Statistics Block

Figure 9-6 shows the format of the statistics block.

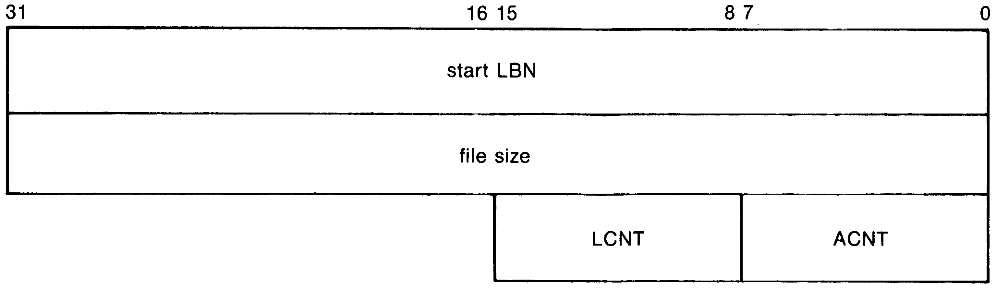


Figure 9-6 ACP QIO Attributes Statistics Block

If the file is contiguous, the first longword contains the LBN of the first block of the file, that is, the starting LBN. For non-contiguous files, this field is zero. The second longword contains the total file size in blocks. The ACNT byte contains the total number of users who are currently accessing the file. The LCNT byte contains the number of write locks on the file.

Both start LBN and file size appear as inverted longwords, that is, the high- and low-order 16 bits are transposed. This is for compatibility with PDP-11 software.

9.3 ACP FUNCTIONS AND ENCODING

All VAX/VMS ACP functions can be expressed using seven function codes and four function modifiers. The function codes are:

- IO$_CREATE -- creates a directory entry or file
- IO$_ACCESS -- searches a directory for a specified file and accesses that file, if found
- IO$_DEACCESS -- deaccesses a file and, if specified, writes the final attributes in the file header
- IO$_MODIFY -- modifies the file attributes and/or file allocation
- IO$_DELETE -- deletes a directory entry and/or file header
- IO$_MOUNT -- informs the ACP when a volume is mounted; requires mount privilege
- IO$_ACPCONTROL -- performs miscellaneous control functions

In addition to the function codes and modifiers, VAX/VMS ACPs take five device/function-dependent arguments, as shown in Figure 9-7.

	31 0
P1:	Address of FIB descriptor
P2:	Address of file name string descriptor (optional)
P3:	Address of word to receive resultant string length (optional)
P4:	Address of resultant string descriptor (optional)
P5:	Address of attribute control block (optional)

Figure 9-7 ACP Device/Function-Dependent Arguments

The first argument, P1, is the address of the File Information Block descriptor. Section 9.1 describes the FIB in detail.

The second argument, P2, is an optional argument used in directory operations. It specifies the address of the descriptor for the file name string to be entered in the directory. The file name itself must be in read/write memory.

The file name string must have the following format:

name.type;version

or

name.typ.version

where name and type may be any combination of alphanumeric characters, plus the asterisk (*) and percent (%) characters. The version must consist of numeric characters or a single asterisk. The total number of alphanumeric and percent characters in the name and type fields must not exceed 9 and 3, respectively. Any number of additional asterisks may be present.

The wild card characters % and * are not legal in IO$_CREATE requests.

If any of the bits FIBM_ALLNAM, FIBM_ALLTYP, and FIB$M_ALLVER are set, then the contents of the corresponding field in the name string is ignored and assumed to be *.

Note that the file name string cannot contain a directory string. The directory is specified by the FIB$W_DID field (see Table 9-1). Only VAX-11 RMS can process directory strings.

Argument P3 is the address of a word to receive the resultant file name string length. The resultant string is not padded. The actual length is returned in P3. P4 is the address of a descriptor for a buffer to receive the resultant file name string. Both these arguments are optional.

The fifth argument, P5, is an optional argument containing the address of the attribute control block. Section 9.2 describes the attribute control block in detail.

Figure 9-8 shows the format for the descriptors.

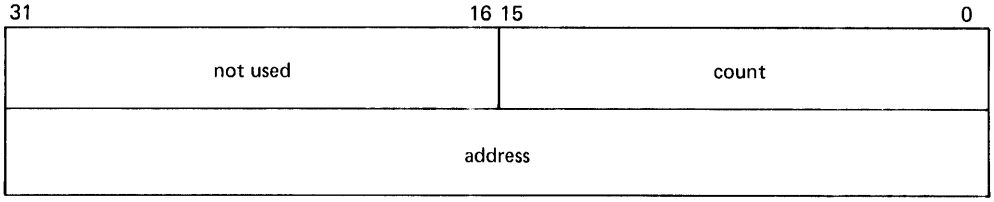


Figure 9-8 ACP Device/Function Argument Descriptor Format

9.3.1 Create File

This virtual I/O function creates a directory entry and/or a file on a disk device, or a file on a magnetic tape device.

The function code is:

IO$_CREATE

The function modifiers are:

- IO$M_CREATE -- creates a file
- IO$M_ACCESS -- opens the file on the user's channel
- IO$M_DELETE -- deletes the file (or marks it for deletion). Applicable only to disk devices.

The device/function-dependent arguments for IO$_CREATE are:

- P1 -- the address of the File Information Block (FIB) descriptor.
- P2 -- the address of the file name string descriptor (optional). The file name is written into the file header; is a directory is specified, this name is entered in the directory. If specified for a magnetic tape file, the name is the name of the created file.
- P3 -- the address of the word that is to receive the length of the resultant file name string (optional).
- P4 -- the address of a descriptor for a buffer that is to receive the resultant file name string (optional).
- P5 -- the address of a list of attribute descriptors (optional). If specified for a disk file, the indicated attributes are written to the file header. If specified for a magnetic tape file, P5 is the address of the descriptor list for the new file.

If the FIB$W_DID field of the FIB is nonzero, the name string is entered in the disk directory specified by the field. If the resultant string descriptor is present, a string representing the full directory entry is returned. If the address of a word to receive the resultant string size is specified, the size, in bytes, of the string is returned. A disk file can also be extended if FIB$M_EXTEND is set. The number of blocks allocated is returned in the second longword of the IOSB.

A disk file header is created if IO$M_CREATE is specified. (The file ID is returned in FIB$W_FID.) If an attribute list is present, the indicated attributes are written to the file header. If IO$M_DELETE is specified, the disk file is marked for deletion. This function modifier may only be used in conjunction with IO$M_CREATE and IO$M_ACCESS.

If IO$M_ACCESS is specified, the disk or magnetic tape file is accessed, that is, opened on the user's channel.

In the name control field (FIB$W_NMCTL) of the FIB, the FIB$M_NEWVER and FIB$M_SUPERSEDE bits function as described in Table 9-1; other flags are ignored. The wild card context field, FIB$L_WCC, is also ignored.

The FIB$L_ACCTL and FIB$W_EXCTL FIB fields are interpreted as described in Table 9-1.

Listed below are the arguments for IO$_CREATE in the order in which they are used. All other areguments are illegal and must be zero:

IO$M_CREATE

FIB$W_DID

FIB$W_NMCTL

Attribute List

IO$M_ACCESS

FIB$L_ACCTL

FIB$M_EXTEND (disk only)

FIB$W_EXCTL (disk only)

FIB$B_ALOPTS

FIB$B_ALALIGN

FIB$B_ALLOC

IO$M_DELETE (disk only)

9.3.2 Access File

This virtual I/O function searches a directory on a disk device, or a magnetic tape, for a specified file and accesses that file if found.

The function code is:

IO$_ACCESS

The function modifiers are:

- IO$M_CREATE -- creates a file
- IO$M_ACCESS -- opens the file on the user's channel

IO$M_CREATE changes the IO$_ACCESS function code to IO$_CREATE if the directory search failed with a "file not found" condition. The function is then re-executed as a CREATE. In that case, the argument interpretations for IO$_CREATE apply, rather than those for IO$_ACCESS. If IO$M_CREATE is specified, the file is accessed. A file must be accessed before it can be read or written.

The device/function-dependent arguments for IO$_ACCESS are:

- P1 -- the address of the File Information Block (FIB) descriptor.

- P2 -- the address of the file name string descriptor (optional). If specified for disks, the directory is searched for this name. If IO$_ACCESS is converted to IO$_CREATE, the name is entered in the directory specified by the FIB. If specified for magnetic tapes, the name identifies the file being sought.

- P3 -- the address of the word that is to receive the length of the resultant file name string (optional).

- P4 -- the address of a descriptor for a buffer that is to receive the resultant file name string (optional).

- P5 -- the address of a list of attribute descriptors (optional). If specified for disks, the indicated attributes are read from the file header. If specified for magnetic tapes, the file attributes are returned to the user.

If the FIB$W_DID field is nonzero, a search is made for the name string indicated in a directory specified by the field. If the resultant string descriptor is present, a string representing the full directory entry is returned. The size of the string is returned if the address of the resultant string size word is present. The file identifier is returned in FIB$W_FID.

Several other FIB fields are used in IO$_ACCESS execution. In the FIB$W_NMCTL field, FIB$M_ALLNAM, FIB$M_ALLTYP, and FIB$M_ALLVER control matching of the name fields. If FIB$M_WILD is set, FIB$W_WCC indicates the position in the directory to resume the search; on return, this field contains the position of the directory entry found. The FIB$L_ACCTL field is interpreted as described in Table 9-1.

If an attribute list is present, the indicated file attributes are read.

Listed below are the arguments for IO$_ACCESS in the order in which they are used. All other arguments are illegal and must be 0.

FIB$W_DID

FIB$W_NMCTL

FIB$W_WCC

IO$M_CREATE

IO$M_ACCESS

FIB$L_ACCTL

Attribute List

(Extend control data is ignored)

9.3.3 Deaccess File

This virtual I/O function deaccesses a file and, if specified, writes final attributes in the file header.

Attributes are written to a disk file if they are present and if the file was accessed for a write operation. (If write access and no attributes are specified, and if FIB$M_DLOCK was set when the file was accessed, the deaccess lock bit is set in the file header, inhibiting further access to that file.)

The function code is:

IO$_DEACCESS

The device/function-dependent arguments for IO$_DEACCESS are:

- P1 -- the address of the File Information Block (FIB) descriptor.
- P5 -- the address of a list of attribute descriptors (optional). If specified, the indicated attributes are written to the file header.

Normally, two arguments are used with IO$_DEACCESS: the attribute list and the FIB$L_ACCTL field (in that order); the FIB$L_ACCTL flag bits are ignored. The FIB$W_FID field can be nonzero. If so, it must match the file identifier of the accessed file. IO$_DEACCESS takes no function modifiers.

A truncate operation can also be performed with IO$_DEACCESS, using the FIB$W_EXCTL and FIB$L_EXVBN arguments described below for IO$_MODIFY. In this case, the arguments are used in the following order:

Attribute List

FIB$W_EXCTL

FIB$L_EXVBN

9.3.4 Modify File

This virtual I/O function modifies the file attributes and/or allocation of a disk file. If used with magnetic tape, modify file is basically a NOP.

The function is:

IO$_MODIFY

The device/function-dependent arguments for IO$_MODIFY are:

- P1 -- the address of the File Information Block (FIB) descriptor.
- P2 -- the address of the file name string descriptor (optional). If specified, the directory is searched for the name.
- P3 -- the address of the word that is to receive the length of the resultant file name string (optional).

- P4 -- the address of a descriptor for a buffer that is to receive the resultant file name string (optional).
- P5 -- the address of a list of attribute descriptors (optional). If specified, the indicated attributes are written to the file header.

An initial search is made for the indicated file string. The search is performed the same way, and with the same consequences, as the IO$_ACCESS search (see Section 9.3.2). If an attribute list is present, attributes are written. The file can be either extended or truncated. If extended (FIB$M_EXTEND is set), the amount is indicated by the extend control data (FIB$L_EXSZ) and the total number of blocks allocated to the file is returned in the second longword of the IOSB. If truncated (FIB$M_TRUNC is set), the file is shortened to the number of blocks specified in FIB$L_EXVBN, minus 1. The file round-up quantity, that is, the resulting file size minus the requested file size, is returned in the second longword of the IOSB.

The FIB$W_EXCTL field is interpreted as described in Table 9-1.

FIB$L_EXVBN and FIB$_EXSZ are used to return the actual starting virtual block number (VBN) and size, respectively, of the area allocated or truncated.

The FIB$W_NMCTL and FIB$L_WCC fields are interpreted as described for IO$_ACCESS. If an attribute list is present, the indicated file attributes are written. IO$_MODIFY takes no function modifiers.

Listed below are the legal arguments for IO$_MODIFY in the order in which they are used. All other arguments are illegal and must be 0.

Attribute List

FIB$W_DID (disk only)

FIB$W_NMCTL (disk only)

FIB$L_WCC (disk only)

FIB$M_EXTEND (disk only)

FIB$L_EXSZ (disk only)

FIB$W_EXCTL (disk only)

FIB$B_ALOPTS (disk only)

FIB$B_ALALIGN (disk only)

FIB$B_ALLOC (disk only)

FIB$M_TRUNC (disk only)

FIB$M_MARKBAD (disk only)

9.3.5 Delete File

This virtual I/O function removes a directory header and/or file header from a disk file.

The function code is:

IO$_DELETE

The function modifier is:

IO$M_DELETE -- deletes the file (or marks it for deletion)

The device/function-dependent arguments for IO$_DELETE are:

- P1 -- the address of the File Information Block (FIB) descriptor.
- P2 -- the address of the file name string descriptor (optional). If specified, the name is removed from the directory specified by the FIB.
- P3 -- the address of the word that is to receive the length of the resultant file name string (optional).
- P4 -- the address of a descriptor for a buffer that is to receive the resultant file name string (optional).

A search is made for the directory entry to be deleted. The search is performed the same way as the IO$_ACCESS search (see Section 9.3.2). The directory entry is then removed. The function modifier (IO$M_DELETE) deletes the file header specified by FIB$W_FID.

Listed below are the legal arguments for IO$_DELETE in the order in which they are used. All other arguments are illegal and must be 0.

FIB$W_DID

FIB$W_NMCTL

FIB$L_WCC

IO$M_DELETE

9.3.6 Mount

This virtual I/O function informs the ACP when a disk or magnetic tape volume is mounted. Mount privilege is required. IO$_MOUNT takes no arguments or function modifiers. Note that this function is only a part of the volume mounting operation. Most of the actual processing is performed by the MOUNT utility.

9.3.7 ACP Control

This virtual I/O function performs miscellaneous control functions, depending on the arguments specified.

The function code is:

IO$_ACPCONTROL

The function modifier is:

IO$M_DMOUNT -- dismounts a volume

The device/function-dependent arguments for IO$_ACPCONTROL are:

- P1 -- the address of the File Information Block (FIB) descriptor.
- P2 -- the address of the file name string descriptor (optional).
- P3 -- the address of the word that is to receive the length of the resultant file name string (optional).
- P4 -- the address of a descriptor for a buffer that is to receive the resultant file name string (optional).
- P5 -- the address of a list of attribute descriptors (optional).

If IO$M_DMOUNT is not set, the FIB control function field (FIB$W_CNTRLFUNC) has one of its field values set. Listed below are the legal arguments for IO$_ACPCONTROL in the order in which they are used; all other arguments are illegal and must be 0:

```
IO$M_DMOUNT

FIB$W_CNTRLFUNC field values:

   FIB$C_REWINDFIL

   FIB$C_POSEND

   FIB$C_NEXTVOL

   FIB$C_SPACE

   FIB$C_REWINDVOL

   FIB$C_ENA_QUOTA

   FIB$C_DSA_QUOTA

   FIB$C_ADD_QUOTA

   FIB$C_EXA_QUOTA

   FIB$C_MOD_QUOTA

   FIB$C_REM_QUOTA

   FIB$C_LOCK_VOL

   FIB$C_UNLK_VOL

FIB$L_CNTRLVAL
```

9.3.7.1 **Disk Quotas** - Disk quota enforcement is enabled by a quota file on the volume, or relative volume 1 if the file is on a volume set. The quota file appears in the volume's master file directory (MFD) under the name QUOTA.SYS;1.

Figure 9-9 shows the format of the block used to transfer quota file data to and from the ACP.

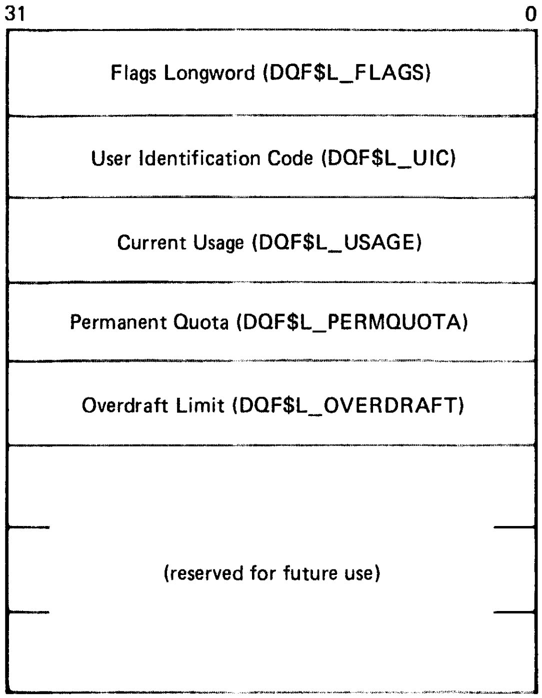

31
0
Flags Longword (DQF$L_FLAGS)
User Identification Code (DQF$L_UIC)
Current Usage (DQF$L_USAGE)
Permanent Quota (DQF$L_PERMQUOTA)
Overdraft Limit (DQF$L_OVERDRAFT)
(reserved for future use)

Figure 9-9 Quota File Transfer Block

In the flags longword, the DQF$V_ACTIVE flag bit is set if this quota file slot is in use.

IO$_ACPCONTROL functions that transfer quota file data between the caller and the ACP use the following device/function-dependent arguments:

- P2 -- the address of a descriptor for a data buffer block that transmits quota file data to the ACP. This block has exactly the same format as a record in the quota file.
- P3 -- the address of a word that returns the data length.
- P4 -- the address of a descriptor for a data block that receives quota file data from the ACP. This block has exactly the same format as a record in the quota file.

Table 9-6 describes the FIB$W_CNTRLFUNC disk quota and lock/unlock bits.

Table 9-6
Disk Quota and Lock/Unlock Bits

Value	Meaning
FIB$C_ENA_QUOTA	Enable the disk quota file. The quota file, if present, is accessed by the ACP and quota enforcement is turned on. To locate the quota file, the directory specified by FIB$W_DID is searched for the name specified in the string given by the P2 argument. The result string and its length are returned in P4 and P5. The quota file must be enabled in order to execute any of the quota file operations listed below. FIB$C_ENA_QUOTA can return the following status values in the IOSB: SS$_NOPRIV No access to quota file SS$_NOQFILE Quota file does not exist SS$_BADQFILE Quota file has bad format SS$_QFACTIVE Quota file is already active (Any of the common error status values, e.g., SS$_BADPARAM and SS$_FCPREADERR, can also be returned)
FIB$C_DSA_QUOTA	Disable the disk quota file. The quota file is deaccessed and quota enforcement is turned off. FIB$C_DSA_QUOTA can return the following status values in the IOSB: SS$_NOPRIV No access to quota file SS$_NOQFILE Quota file does not exist SS$_QFNOTACT Quota file is not active (Any of the common error status values, e.g., SS$_BADPARAM and SS$_FCPREADERR can also be returned)
FIB$C_ADD_QUOTA	Add an entry to the disk quota file, using the UIC and quota specified in the P2 argument block. FIB$C_ADD_QUOTA requires write access to the quota file. The following status values can be returned in the IOSB: SS$_NOPRIV No access to the quota file SS$_NOQFILE Quota file does not exist, or is not enabled SS$_DUPDSKQUOTA Quota entry for UIC already exists (Any of the common error status values, e.g., SS$_BADPARAM and SS$_FCPREADERR can also be returned)

(continued on next page)

Table 9-6
Disk Quota and Lock/Unlock Bits

Value	Meaning
FIB$C_EXA_QUOTA	Examine a disk quota file entry. The entry whose UIC is specified in the P2 argument block is returned in the P4 argument block, and its length is returned in the P3 argument word. Using two flags in FIB$L_CNTRLVAL, it is possible to wild card through the quota file. (The ACP maintains position context in FIB$L_WCC, and each examine call returns the next matching entry.) The two flags are: FIB$M_ALL_MEM Match all UIC members FIB$M_ALL_GRP Match all UIC groups Read access is required to examine all entries not belonging to the user. FIB$C_EXA_QUOTA can return the following status values in the IOSB: SS$_NOPRIV No access to quota file SS$_NOQFILE Quota file does not exist, or is not enabled SS$_NODISKQUOTA Specified quota file entry does not exist (Any of the common error status values, e.g., SS$_BADPARAM and SS$_FCPREADERR can also be returned)
FIB$C_MOD_QUOTA	Modify a disk quota file entry. The quota file entry specified by the UIC in the P2 argument block is modified according to the values in the block, as controlled by two flags in FIB$L_CNTRLVAL: FIB$M_MOD_PERM Change the permanent quota FIB$M_MOD_USE Change the usage data The usage data can be changed only if the volume is locked by FIB$C_LOCK_VOL (see below). FIB$C_MOD_QUOTA requires write access. The following status values can be returned in the IOSB: SS$_NOPRIV No access to quota file SS$_NOQFILE Quota file does not exist, or is not enabled SS$_NODISKQUOTA Specified quota file entry does not exist SS$_OVRDSKQUOTA Usage is greater than quota SS$_ACCONFLICT Volume is not locked (usage change) (Any of the common error status values, e.g., SS$_BADPARAM and SS$_FCPREADERR can also be returned)

(continued on next page)

Table 9-6 (Cont.)
Disk Quota and Lock/Unlock Bits

Value	Meaning
FIB$C_REM_QUOTA	Remove a disk quota file entry whose UIC is specified in the P2 argument block. FIB$C_REM_QUOTA requires write access to the quota file. The following status values can be returned in the IOSB: SS$_NOPRIV No access to quota file SS$_NOQFILE Quota file does not exist, or is not enabled SS$_NODISKQUOTA Specified quota file entry does not exist SS$_OVRDSKQUOTA Usage is non-zero (Any of the common error status values, e.g., SS$_BADPARAM and SS$_FCPREADERR can also be returned)
FIB$C_LOCK_VOL	Allocation lock the volume; operations which change the file structure are not permitted. This function must be executed prior to rebuilding the quota file. To issue this function, the user must either have a system UIC or SYSPRV privilege, or be the owner of the volume. FIB$C_LOCK_VOL can return the following status values in the IOSB: SS$_NOPRIV No access to volume (Any of the common error status values, e.g., SS$_BADPARAM and SS$_FCPREADERR can also be returned)
FIB$C_UNLK_VOL	Unlock the volume. Cancels FIB$C_LOCK_VOL. To issue this function, the user must either have a system UIC or SYSPRV privilege, or be the owner of the volume. FIB$C_UNLK_VOL can return the following status values in the IOSB: SS$_NOPRIV No access to volume (Any of the common error status values, e.g., SS$_BADPARAM and SS$_FCPREADERR can also be returned)

9.4 I/O STATUS BLOCK

Figure 9-10 shows the I/O status block (IOSB) for ACP QIO functions. Table 9-7 lists the status returns for these functions.

The file ACP returns a completion status in the first longword of the IOSB. In an extend operation, the second longword is used to return the number of blocks allocated to the file. If a contiguous extend operation (FIB$M_ALCON) fails, the second longword is used to return the size of the file after truncation.

Values returned in the IOSB are most useful during operations in compatibility mode. When executing programs in the native mode, the user should use the values returned in FIB locations.

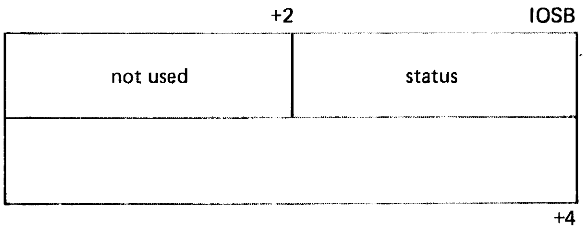


Figure 9-10 IOSB Contents - ACP QIO Functions

If an extend operation (including CREATE) was performed, IOSB+4 contains the number of blocks allocated, or the largest available contiguous space if a contiguous extend operation failed. If a truncate operation was performed, IOSB+4 contains the number of blocks added to the file size to reach the next cluster boundary.

Table 9-7
ACP QIO Status Returns

Status	Meaning
SS$_ACCONFLICT	Access mode conflict. Requested access mode conflicted with existing file accesses, for example, an attempt to open a file for a write when the file is write locked.
SS$_ACPVAFUL	The magnetic tape ACP's virtual address space is full. Since each volume set has a virtual page assigned to it, additional volume sets cannot be handled. Corrective action consists of starting a different ACP using the unique switch in MOUNT.
SS$_BADATTRIB	Invalid attribute code or size specified in read or write attribute list.
SS$_BADCHKSUM	Invalid checksum in the file header.
SS$_BADFILEHDR	Invalid file header, for example, structure is inconsistent or the storage map indicates blocks are marked free.
SS$_BADFILENAME	Invalid syntax in file name string. The string contains illegal characters, or is larger than 9 characters.
SS$_BADFILEVER	Invalid file version number, that is, a number greater than 32767.
SS$_BADIRECTORY	Invalid directory file. The file is not a directory or the file contains invalid data.
SS$_BADPARAM	Invalid parameter list. For example, the FIB contains options not applicable to this function.

(continued on next page)

Table 9-7 (Cont.)
ACP QIO Status Returns

Status	Meaning
SS$_BADQFILE	Bad quota file. The quota file has an invalid format.
SS$_BLOCKCNTERR	Block count error. The number of blocks read differs from the number of blocks recorded in the trailer labels. There is a possibility that a record was skipped or an extra noise record was read.
SS$_CREATED	File created by an ACCESS function with a CREATE function modifier. (A success status return.)
SS$_DEVICEFULL	Device full. No free blocks are available on the device or the number of contiguous blocks specified in a contiguous request is not available.
SS$_DIRFULL	Directory if full. An error occurred while creating a disk file because the directory specified is full and cannot catalog any more entries. A directory is limited to 1024 blocks.
SS$_DUPDSKQUOTA	Duplicate disk quota. Another quota entry for UIC already exists.
SS$_DUPFILENAME	Duplicate file name. Another directory entry with the same name, type, and version already exists.
SS$_ENDOFFILE	End-of-file. End of allocated space encountered in a virtual I/O operation or an attempted truncation.
SS$_FCPREADERR	FCP read error. An I/O error occurred when file structure data, for example, a directory, was read.
SS$_FCPREWINDERR	File process rewind error. An I/O error occurred when rewinding a volume.
SS$_FCPSPACERR	File process space error. An I/O error occurred when spacing within a file or spacing files.
SS$_FCPWRITERR	FCP write error. An I/O error occurred when file structure data, for example, a directory, was written.
SS$_FILELOCKED	File deaccess locked. Attempted to access a locked file. A file becomes locked when it is accessed with FIB$M_DLOCK set and then deaccessed without writing attributes.

(continued on next page)

Table 9-7 (Cont.)
ACP QIO Status Returns

Status	Meaning
SS$_FILENUMCHK	File identifier number check. The index file contains invalid data.
SS$_FILESEQCHK	File identifier sequence check. A directory entry points to a file that has been deleted.
SS$_FILESTRUCT	Unsupported file structure. The file structure on the accessed volume is not compatible with the ACP. For example, an attempt was made to use a structure level 2 ACP with a structure level 1 disk.
SS$_FILNOTEXP	File not expired. A magnetic tape file that has not expired cannot be written over unless the override expiration qualifier was specified to MOUNT.
SS$_HEADERFULL	File header map area is full and header extension is inhibited. This can occur on a volume's index file in a CREATE operation.
SS$_IDXFILEFULL	Volume index file is full. The maximum number of files specified at initialization time has been reached.
SS$_ILLCNTRFUNC	Illegal control function. An illegal function is specified for IO$_ACPCONTROL.
SS$_NODISKQUOTA	No disk quota. The specified quota file entry does not exist.
SS$_NOMOREFILES	No more files exist which match the given wild card in a file specification string. At least one file was found, that is, one match was made.
SS$_NOPRIV	No privilege. Volume or file protection will not allow the requested operation.
SS$_NOQFILE	No quota file. The quota file does not exist.
SS$_NOSUCHFILE	No such file. No file with the given file name or file identifier exists. Can be caused by a directory entry that points to a file that has been deleted.
SS$_NOTAPEOP	No tape operator. There is no tape operator and a need to communicate with one exists, for example, the next volume in a volume set must be mounted.
SS$_NOTLABELMT	Magnetic tape not labeled. A request to read a magnetic tape failed because the tape does not have standard labels.

(continued on next page)

Table 9-7 (Cont.)
ACP QIO Status Returns

Status	Meaning
SS$_OVRDSKQUOTA	Over disk quota. Disk usage exceeds quota. (A success status return.)
SS$_QFACTIVE	Quota file is already active.
SS$_QFNOTACT	Quota file not active.
SS$_SUPERSEDE	An existing file of the same name, type, and version has been deleted by a CREATE function. (A success status return.)
SS$_TAPEPOSLOST	Magnetic tape position lost.
SS$_TOOMANYVER	Too many versions. The maximum number of file versions already exists. All are higher versions than the one being created.
SS$_WRTLCK	The device is software write locked or the hardware write lock switch on the drive is set.

CHAPTER 10

LABORATORY PERIPHERAL ACCELERATOR DRIVER

This chapter describes the use of the VAX/VMS Laboratory Peripheral Accelerator (LPA11-K) driver and the high-level language procedure library that interfaces with the LPA11-K driver. The procedure library is implemented with callable assembly language routines that translate arguments into the format required by the LPA11-K driver and handle buffer chaining operations. Routines for microcode loading and device initialization are also described.

This chapter is written with the understanding that the reader has access to a copy of the LPA11-K Laboratory Peripheral Accelerator User's Guide.

10.1 SUPPORTED DEVICE

The LPA11-K is a peripheral device that controls analog-to-digital (A/D) and digital-to-analog (D/A) converters, digital I/O registers, and real-time clocks. It is connected to the VAX-11 processor through the UNIBUS Adapter (UBA).

The LPA11-K is a fast, flexible, and easy to use microprocessor subsystem that is designed for applications requiring concurrent data acquisition and data reduction at high rates. The LPA11-K allows aggregate analog input and output rates up to 150,000 samples per second. The maximum aggregate digital input and output rate is 15,000 samples per second.

Table 10-1 lists the useful minimum and maximum LPA11-K configurations supported by VAX/VMS.

10.1.1 LPA11-K Modes of Operation

The LPA11-K operates in two distinct modes: dedicated and multirequest.

In dedicated mode only one user, that is, one request, can be active at a time, and only analog I/O data transfers are supported. Up to two A/D converters can be controlled simultaneously. One D/A converter can be controlled at a time. Sampling is initiated either by an overflow of the real-time clock or by an externally supplied signal. Dedicated mode provides sampling rates of up to 150,000 samples per second.

Table 10-1
Minimum and Maximum Configurations per LPA11-K

Minimum	Maximum
1 - DD11-Cx or Dx Backplane	2 - DD11-Cx or Dx Backplanes
1 - KW11-K Real Time Clock	1 - KW11-K Real Time Clock
One of the following:	2 - AD11-K A/D Converters
AD11-K A/D Converter AA11-K A/D Converter DR11-K Digital I/O Register	2 - AM11-K Multiplexers for AD11-K Converters
	1 - AA11-K D/A Converter
	5 - DR11-K Digital I/O Registers

In multirequest mode, sampling from all the devices listed in Table 10-1 is supported. The LPA11-K operates like a multicontroller device; up to eight requests (from one through eight users) can be active simultaneously. The sampling rate for each user is a multiple of the common real-time clock rate. Independent rates can be maintained for each user. Both the sampling rate and the device type are specified as part of each data transfer request. Multirequest mode provides a maximum aggregate sampling rate of 15,000 samples per second.

10.1.2 Errors

The LPA11-K returns three classes of errors:

1. Errors associated with the issuance of a new LPA11-K command (SS$_DEVCMDERR).

2. Errors associated with an active data transfer request (SS$_DEVREQERR).

3. Fatal hardware errors which affect all LPA11-K activity (SS$_CTRLERR).

Appendix A of the LPA11-K Laboratory Peripheral Accelerator User's Guide lists these three classes of errors and the specific error codes for each class. The LPA11-K aborts all active requests if any of the following conditions occur:

- Power failure

- Device timeout

- Fatal error

Power failure is reported to any active users when power is recovered.

Device timeouts are monitored only when a new command is issued. For data transfers, the time between buffer full interrupts is not defined. Thus, no timeout errors are reported on a buffer to buffer basis.

If a required resource is not available to a process, an error message is returned immediately. The driver does not place the process in the resource wait mode.

10.2 SUPPORTING SOFTWARE

The LPA11-K is supported by a device driver, a high-level language procedure library of support routines, and routines for microcode loading and device initialization. All data transfer algorithms for the laboratory data acquisition I/O devices are accomplished by the LPA11-K. The only purpose for the system software and support routines is to provide a control path for synchronizing the use of buffers, specifying requests, and starting and stopping requests.

The LPA11-K driver and the associated I/O interface have the following features:

- They permit multiple LPA11-K subsystems on a single UBA.
- They operate as an integral part of the VAX/VMS operating system.
- They can be loaded on an operating VAX/VMS system without relinking the executive.
- They handle I/O requests, function dispatching, UBA map allocation, interrupts, and error-reporting for multiple LPA11-K subsystems.
- The LPA11-K functions as a multibuffered device. Up to eight buffer areas can be defined per request. Up to eight requests can be handled simultaneously. Buffer areas can be reused after the data they contain is processed.
- Since the LPA11-K chains buffer areas automatically, a start data transfer request can transfer an infinite and continuous amount of data.
- Multiple ASTs are dynamically queued by the driver to indicate when a buffer has been filled (the data is available for processing) or emptied (the buffer is available for new data).

The high-level language support routines have the following features:

- They translate arguments provided in the high-level language calls into the format required for the Queue I/O interface.
- They provide a buffer chaining capability for a multibuffering environment by maintaining queues of used, in use, and available buffers.
- They adhere to all VAX/VMS conventions for calling sequences, use of shareable resources, and reentrancy.
- They can be part of a resident global library, or be linked into a process image as needed.

The routines for microcode loading and device initialization have the following features:

- They execute, as separate processes, images which issue I/O requests. These I/O requests initiate microcode image loading, start the LPA11-K subsystem, and automatically configure the peripheral devices on the LPA11-K internal I/O bus.
- They can be executed by user or operator request.
- They can be executed at the request of other processes.
- They can be executed automatically when the system is initialized and on power recovery.

Figure 10-1 shows the relationship of the supporting software to the LPA11-K.

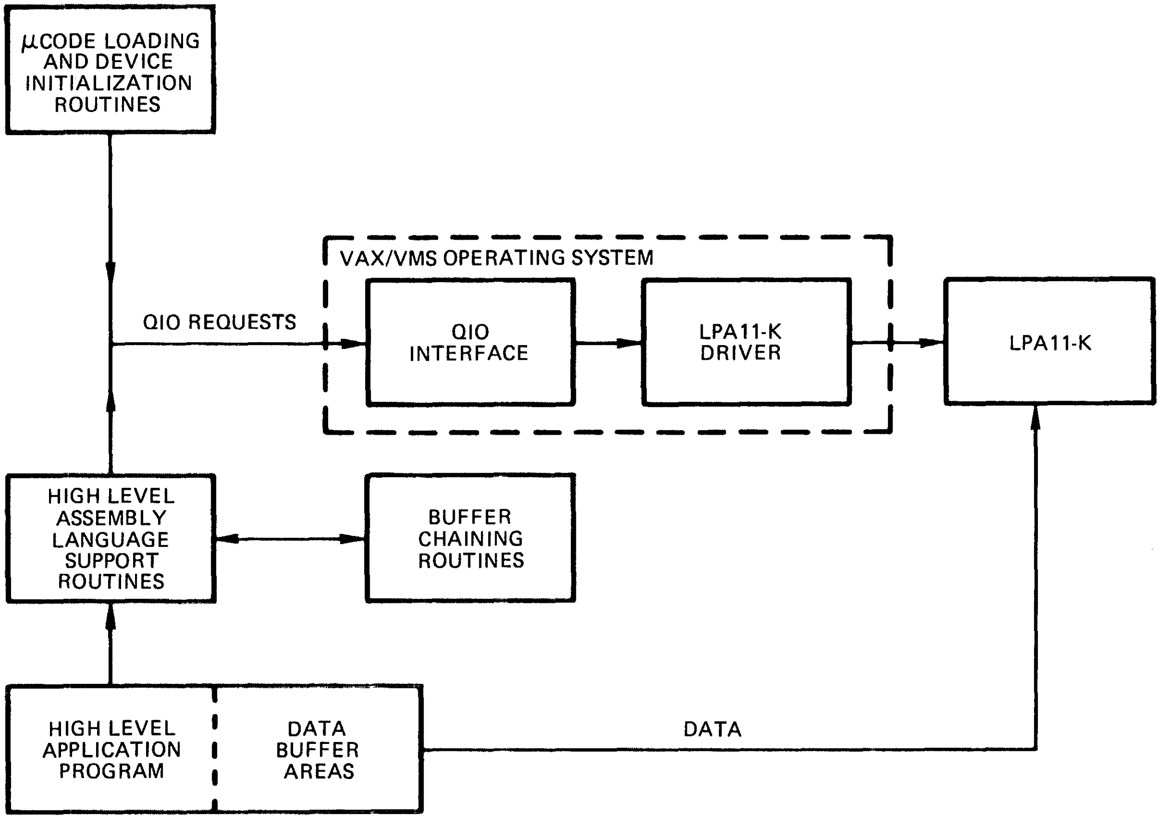

μCODE LOADING AND DEVICE INITIALIZATION ROUTINES
VAX/VMS OPERATING SYSTEM
QIO REQUESTS
QIO INTERFACE
LPA11-K DRIVER
LPA11-K
HIGH LEVEL ASSEMBLY LANGUAGE SUPPORT ROUTINES
BUFFER CHAINING ROUTINES
HIGH LEVEL APPLICATION PROGRAM
DATA BUFFER AREAS
DATA

Figure 10-1 Relationship of Supporting Software to LPA11-K

10.3 DEVICE INFORMATION

Users can obtain information on all peripheral data acquisition devices on the LPA11-K internal I/O bus by using the $GETCHN and $GETDEV system services (see Section 1.10). The LPA11-K-specific information is returned in the first three longwords of a user-specified buffer, as shown in Figure 10-2 (Figure 1-9 shows the entire buffer).

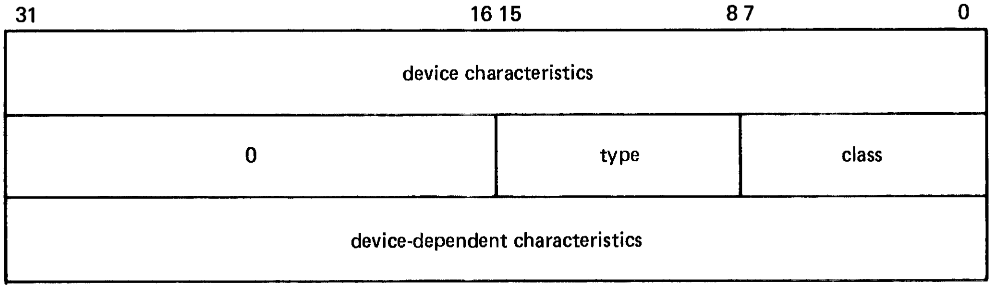


Figure 10-2 LPA11-K Information

The first longword contains device-independent information. The second and third longwords contain device-dependent data.

Table 10-2 lists the device-independent characteristics returned in the first longword.

Table 10-2
Device-independent Characteristics

Dynamic Bits[1] (Conditionally Set)	Meaning
DEV$M_AVL	Device is online and available
Static Bits[1] (Always Set)	
DEV$M_IDV	Input device
DEV$M_ODV	Output device
DEV$M_RTM	Real-time device
DEV$M_SHR	Device is shareable

[1] Defined by the $DEVDEF macro.

The second longword contains information on the device class and type. The device class for the LPA11-K is DC$_REALTIME and the device type is DT$_LPA11. The $LADEF macro defines these values. Buffer size is not applicable to the LPA11-K; this word is 0.

The third longword contains LPA11-K characteristics, that is, device-dependent data. LPA11-K characteristics are set by the set clock, initialize, and load microcode I/O functions to any one of, or a combination of, the values listed in Table 10-3.

Table 10-3
Device-Dependent Characteristics

Field[1]	Meaning
LA$M_MCVALID LA$S_MCVALID LA$V_MCVALID	The load microcode I/O function (IO$_LOADMCODE) was performed successfully. LA$M_MCVALID is set by IO$_LOADMCODE. Each microword is verified by reading it back and comparing it with the specified value. LA$M_MCVALID is cleared if there is no match.
LA$V_MCTYPE LA$S_MCTYPE	The microcode type, set by the load microcode I/O function (IO$_LOADMCODE), is one of the following values: LA$K_MRMCODE = microcode type is in multirequest mode LA$K_ADMCODE = microcode type is in dedicated A/D mode LA$K_DAMCODE = microcode type is in dedicated D/A mode
LA$V_CONFIG LA$S_CONFIG	The bit positions, set by the initialize I/O function (IO$_INITIALIZE), for the peripheral data acquisition devices on the LPA11-K internal I/O bus are one or more of the following: LA$V_CLOCKA = Clock A LA$M_CLOCKA LA$V_CLOCKB = Clock B LA$M_CLOCKB LA$V_AD1 = A/D device 1 LA$M_AD1 LA$V_AD2 = A/D device 2 LA$M_AD2 LA$V_DA = D/A device 1 LA$M_DA LA$V_DIO1 = Digital I/O Buffer 1 LA$M_DIO1 LA$V_DIO2 = Digital I/O Buffer 2 LA$M_DIO2 LA$V_DIO3 = Digital I/O Buffer 3 LA$M_DIO3 LA$V_DIO4 = Digital I/O Buffer 4 LA$M_DIO4 LA$V_DIO5 = Digital I/O Buffer 5 LA$M_DIO5

[1] Values defined by the $LADEF macro.

(continued on next page)

Table 10-3 (Cont.)
Device-Dependent Characteristics

Field[1]	Meaning
LA$V_RATE LA$S_RATE	The Clock A rate, set by the set clock function (IO$_SETCLOCK), is one of the following values: 0 = Stopped 1 = 1 MHz 2 = 100 kHz 3 = 10 kHz 4 = 1 kHz 5 = 100 Hz 6 = Schmidt trigger 7 = Line frequency
LA$V_PRESET LA$S_PRESET	The Clock A preset value set by the set clock function (IO$_SETCLOCK). (The value is in the range 0 through 65,535 - in 2's complement form.) The clock rate divided by the clock preset value yields the clock overflow rate.

[1] Values defined by the $LADEF macro.

10.4 LPA11-K I/O FUNCTION CODES

The LPA11-K I/O functions are:

1. Load microcode into the LPA11-K.

2. Start the LPA11-K microprocessor.

3. Initialize the LPA11-K subsystem.

4. Set the LPA11-K real-time clock rate.

5. Start a data transfer request.

The Cancel I/O on Channel ($CANCEL) system service is used to abort data transfers.

10.4.1 Load Microcode

This I/O function resets the LPA11-K and loads an image of LPA11-K microcode. Physical I/O privilege is required. VAX/VMS defines a single function code:

IO$_LOADMCODE - load microcode

The load microcode function takes three device/function dependent arguments:

- P1 = the starting virtual address of the microcode image that is to be loaded into the LPA11-K
- P2 = the number of bytes (usually 2048) that are to be loaded
- P3 = the starting microprogram address (usually 0) in the LPA11-K that is to receive the microcode

If any data transfer requests are active at the time a load microcode request is issued, the load request is rejected and SS$_DEVACTIVE is returned in the I/O status block.

Each microword is verified by comparing it with the specified value in memory. If all words match, that is, if the microcode was loaded successfully, the driver sets the microcode valid bit (LA$V_MCVALID) in the device-dependent characteristics longword (see Table 10-3). If there is no match, SS$_DATACHECK is returned in the I/O status block and LA$V_MCVALID is cleared to indicate that the microcode was not properly loaded. If the microcode was loaded successfully, the driver stores one of the microcode type values (LAK_MRCODE, LAK_ADCODE, or LA$K_DAMCODE) in the characteristics longword.

After a load microcode function is completed, the second word of the I/O status block contains the number of bytes loaded.

In addition to SS$_DATACHECK, IO$_LOADMCODE can return SS$_DEVACTIVE in the I/O status block.

10.4.2 Start Microprocessor

This I/O function resets the LPA11-K and starts (or restarts) the LPA11-K microprocessor. Physical I/O privilege is required. VAX/VMS defines a single function code:

IO$_STARTMPROC - start microprocessor

This function code takes no device/function-dependent arguments.

The start microprocessor function can return five error codes in the I/O status block: SS$_DEVACTIVE, SS$_MCNOTVALID, SS$_CTRLERR, SS$_POWERFAIL, and SS$_TIMEOUT (see Section 10.6).

10.4.3 Initialize LPA11-K

This I/O function issues a subsystem initialize command to the LPA11-K. This command specifies LPA11-K laboratory I/O device addresses and other table information for the subsystem. It is issued only once after restarting the subsystem and before any other LPA11-K command is given. Physical I/O privilege is required. VAX/VMS defines a single function code:

IO$_INITIALIZE - initialize LPA11-K

The initialize LPA11-K function takes two device/function-dependent arguments:

- P1 = the starting, word-aligned, virtual address of the Initialize Command Table in the user process. This table is read once by the LPA11-K during the execution of the initialize command. See the LPA11-K Laboratory Peripheral Accelerator User's Guide for additional information.

- P2 = length of the initialize command buffer (always 278 bytes)

If the initialize function is completed successfully, the appropriate device configuration values are set in the device-dependent characteristics longword (see Table 10-3).

The initialize function can return ten error codes in the I/O status block: SS$_IVMODE, SS$_INCLENGTH, SS$_BUFNOTALIGN, SS$_CTRLERR, SS$_DEVCMDERR, SS$_CANCEL, SS$_INSFMAPREG, SS$_MCNOTVALID, SS$_POWERFAIL, and SS$_TIMEOUT (see Section 10.6).

If a device specified in the Initialize Command Table is not in the LPA11-K configuration, an error condition (SS$_DEVCMDERR) occurs and the address of the first device not found is returned in the LPA11-K maintenance status register (see Section 10.6). A program can use this characteristic to poll the LPA11-K and determine the current device configuration.

10.4.4 Set Clock

This virtual function issues a clock control command to the LPA11-K. The clock control command specifies information necessary to start, stop, or change the sample rate at which the real-time clock runs on the LPA11-K subsystem.

If the LPA11-K has more than one user, caution should be exercised when the clock rate is changed. In multirequest mode, a change in the clock rate will affect all users.

VAX/VMS defines a single function code:

IO$_SETCLOCK - set clock

The set clock function takes three device/function-dependent arguments:

- P2 = mode of operation. VAX/VMS defines the following clock start mode word (hexadecimal) values:

 1 = KW11-K Clock A
 11 = KW11-K Clock B

- P3 = clock control and status. VAX/VMS defines the following clock status word (hexadecimal) values:

0	= stop clock
143	= 1 MHz clock rate
145	= 100 kHz clock rate
147	= 10 kHz clock rate
149	= 1 kHz clock rate
14B	= 100 Hz clock rate
14D	= clock rate is Schmidt trigger 1
14F	= clock rate is line frequency

- P4 = the 2's complement of the real-time clock preset value. The range is 16 bits for the KW11-K Clock A and 8 bits for the KW11-K Clock B.

The LPA11-K Laboratory Peripheral Accelerator User's Guide describes the clock start mode word and the clock status word in greater detail.

If the set clock function is completed successfully for Clock A, the clock rate and preset values are stored in the device-dependent characteristics longword (see Table 10-3).

The set clock function can return six error codes in the I/O status block: SS$_CTRLERR, SS$_DEVCMDERR, SS$_CANCEL, SS$_MCNOTVALID, SS$_POWERFAIL, and SS$_TIMEOUT (see Section 10.6).

10.4.5 Start Data Transfer Request

This virtual I/O function issues a Data Transfer Start command that specifies the buffer addresses, sample mode, and sample parameters used by the LPA11-K. This information is passed to the Data Transfer Command Table. VAX/VMS defines a single function code:

IO$_STARTDATA - start data transfer request

The start data transfer request function takes one function modifier:

IO$M_SETEVF - set event flag

The start data transfer request function takes four device/function-dependent arguments:

- P1 = the starting virtual address of the Data Transfer Command Table in the user's process

- P2 = the length in bytes (always 40) of the Data Transfer Command Table

- P3 = the AST address of the normal buffer completion AST routine (optional)

- P4 = the AST address of the buffer overrun completion AST routine (optional). Only used when the buffer overrun bit (LA$M_BFROVRN) is set, that is, a buffer overrun condition is classified as a non-fatal error.

A buffer overrun condition is not the same as a data overrun condition. The LPA11-K fetches data from, or stores data in, memory. If data cannot be fetched quickly enough, for example, when there is too much UNIBUS activity, a data underrun condition occurs. If data cannot be stored quickly enough, a data overrun condition occurs. After each buffer has been filled or emptied, the LPA11-K obtains the index number of the next buffer to process from the User Status Word (USW). (See Section 2.5 of the LPA11-K Laboratory Peripheral Accelerator User's Guide). A buffer overrun condition occurs if the LPA11-K fills or empties buffers faster than the application program can supply new buffers. For example, buffer overrun can occur when the sampling rate is too high, the buffers are too small, or the system load is too heavy.

The LPA11-K driver accesses the 10-longword Data Transfer Command Table, shown in Figure 10-3, when the Data Transfer Start command is processed. After the command is accepted and data transfers have begun, the driver makes no further access to the table.

31 24	23 16	15 8	7 0
	highest available buffer and buffer overrun bit	mode	
user status word address			
overall data buffer length			
overall data buffer address			
random channel list length			
random channel list address			
channel increment	start channel number	delay	
dwell		number of channels	
digital trigger mask		event mark channel	digital trigger channel
		event mark mask	

Figure 10-3 Data Transfer Command Table

In the first longword of the Data Transfer Command Table, the first two bytes contain the LPA11-K start data transfer request mode word. (Section 3.4.1 of the LPA11-K Laboratory Peripheral Accelerator User's Guide describes the functions of this word.)

The third byte contains the number (0-7) of the highest buffer available and the buffer overrun flag bit (bit 23; values: LA$M_BFROVRN and LA$V_BFROVRN). If this bit is set, a buffer overrun condition is a nonfatal error.

The second longword contains the User Status Word address (see Section 3.4.3 of the LPA11-K Laboratory Peripheral Accelerator User's Guide). This virtual address points to a 2-byte area in the user process space, and must be word-aligned.

The third longword contains the size (in bytes) of the overall buffer area. The virtual address in the fourth longword is the beginning address of this area. This address must be longword-aligned. The overall buffer area contains a specified number of buffers (the number of the highest available buffer specified in the first longword plus one). Individual buffers are subject to length restrictions: in multirequest mode the length must be in multiples of two bytes; in dedicated mode the length must be in multiples of four bytes. All data buffers are virtually contiguous for each data transfer request.

The fifth and sixth longwords contain the Random Channel List (RCL) length and address, respectively. The RCL address must be word-aligned. The last word in the RCL must have bit 15 set. (See Section 3.4.6 of the LPA11-K Laboratory Peripheral Accelerator User's Guide for additional information on the RCL.)

The seventh through tenth longwords contain LPA11-K-specific sample parameters. The driver passes these parameters directly to the LPA11-K. (See Sections 3.4.7 through 3.4.12 of the LPA11-K Laboratory Peripheral Accelerator User's Guide for a detailed description of their functions.)

The start data transfer request function can return 15 error codes in the I/O status block: SS$_INCLENGTH, SS$_BUFNOTALIGN, SS$_DEVCMDERR, SS$_CTRLERR, SS$_DEVREQERR, SS$_ABORT, SS$_CANCEL, SS$_EXQUOTA, SS$_INSFBUFDP, SS$_INSFMAPREG, SS$_INSFMEM, SS$_MCNOTVALID, SS$_PARITY, SS$_POWERFAIL, and SS$_TIMEOUT (see Section 10.6).

Data buffers are chained and reused as the LPA11-K and the user process dispose of the data. As each buffer is filled or emptied, the LPA11-K driver notifies the application process by either setting the event flag specified by the QIO request efn argument or queueing an AST. Since buffer use is a continuing process, the event flag is set or the AST is queued a number of times. The user process must clear the event flag (or receive the AST), process the data, and specify the next buffer for the LPA11-K to use.

If the set event flag function modifier (IO$M_SETEVF) is specified, the event flag is set repeatedly: when the data transfer request is started, on each buffer completion, and when the request completes. If IO$M_SETEVF is not specified, the event flag is set only when the request completes.

ASTs are preferred over event flags for synchronizing a program with the LPA11-K because AST delivery is a queued process while setting of event flags is not. If only event flags are used, it is possible to lose buffer status.

Three AST addresses can be specified. For normal data buffer transactions the AST address specified in the P3 argument is used. If the buffer overrun bit in the Data Transfer Command Table is set and an overrun condition occurs, the AST address specified in the P4 argument is used. The AST address specified in the astadr argument of the QIO request is used when the entire data transfer request is completed. The astprm argument specified in the QIO request is passed to all three AST routines.

If insufficient dynamic memory is available to allocate an AST block, an error (SS$_INSFMEM) is returned. If the user does not have sufficient AST quota remaining to allocate an AST block, an error (SS$_EXQUOTA) is returned. In either case, the request is stopped. Normally, there are never more than three outstanding ASTs per LPA11-K request.

10.4.6 LPA11-K Data Transfer Stop Command

The Cancel I/O on Channel ($CANCEL) system service is used to abort data transfers for a particular process. When the LPA11-K driver receives a $CANCEL request, a Data Transfer Stop command is issued to the LPA11-K.

The normal way to stop a data transfer is to set bit 14 of the User Status Word. If this bit is set, the transfer stops at the end of the next buffer transaction (see Section 2.5 of the LPA11-K Laboratory Peripheral Accelerator User's Guide).

10.5 HIGH LEVEL LANGUAGE INTERFACE

VAX/VMS supports several program-callable procedures that provide access to the LPA11-K. The formats of these calls are documented here for VAX-11 FORTRAN users. VAX-11 MACRO users must set up a standard VAX/VMS argument block and issue the standard procedure CALL. (Optionally, VAX-11 MACRO users can access the LPA11-K directly through the use of the device-specific Queue I/O functions described in Section 10.4.) Users of other high-level languages must specify the proper subroutine or procedure invocation.

10.5.1 High-level Language Support Routines

VAX/VMS provides 20 high-level language procedures for the LPA11-K. These procedures are divided into four classes. Table 10-4 lists the VAX-11 procedures for the LPA11-K.

Table 10-4
VAX-11 Procedures for the LPA11-K

Class	Subroutine	Function
Sweep Control	LPA$ADSWP	Start A/D converter sweep
	LPA$DASWP	Start D/A converter sweep
	LPA$DISWP	Start digital input sweep
	LPA$DOSWP	Start digital output sweep
	LPA$LAMSKS	Specify LPA11-K controller and digital mask words
	LPA$SETADC	Specify channel select parameters
	LPA$SETIBF	Specify buffer parameters
	LPA$STPSWP	Stop sweep
Clock control	LPA$CLOCKA	Set Clock A rate
	LPA$CLOCKB	Set Clock B rate
	LPA$XRATE	Compute clock rate and preset value
Data Buffer Control	LPA$IBFSTS	Return buffer status
	LPA$IGTBUF	Return next available buffer
	LPA$INXTBF	Alter buffer order
	LPA$IWTBUF	Return next buffer or wait
	LPA$RLSBUF	Release buffer to LPA11-K
	LPA$RMVBUF	Remove buffer from device queue
Miscellaneous	LPA$CVADF	Convert A/D input to floating point
	LPA$FLT16	Convert unsigned integer to floating point
	LPA$LOADMC	Load microcode and initialize LPA11-K

10.5.1.1 **Buffer Queue Control** - This section is provided for informational purposes only. Normally, the user does not need to be concerned with the details of buffer queues.

Buffer queue control for data transfers by LPA11-K subroutines involves the use of three queues:

- Device queue (DVQ)
- User queue (USQ)
- In-use queue (IUQ)

Each data transfer request can specify from one through eight data buffer areas. The user specifies these buffers by address. During execution of the request, the LPA11-K assigns an index from 0 through 7 when a buffer is referenced.

The DVQ contains the indices of all the buffers that the user has released, that is, made available to be filled or emptied by the LPA11-K. For output functions (D/A and digital output), these buffers contain data to be output by the LPA11-K. For input functions (A/D and digital input), these buffers are empty and waiting to be filled by the LPA11-K.

The USQ contains the indices of all buffers that are waiting to be returned to the user. The LPA$IWTBUF and LPA$IGTBUF calls are used to return the index of the next buffer in the USQ. For output functions (D/A and digital output), these buffers are empty and waiting to be filled by the application program. For input functions (A/D and digital input), these buffers contain data to be processed by the application program.

The IUQ contains the indices of all buffers that are currently being processed by the LPA11-K. Normally, the IUQ contains the indices of two buffers:

- The buffer currently being filled or emptied by the LPA11-K
- The next buffer to be filled or emptied by the LPA11-K. This is the buffer specified by the next buffer index field in the User Status Word.

Because the LPA11-K driver requires that at least one buffer be ready when the input or output sweep is started, the user must call LPA$RLSBUF before the sweep is initiated.

Figure 10-4 shows the flow between the buffer queues.

10.5.1.2 **Subroutine Argument Usage** - Table 10-5 describes the general use of the subroutine arguments. The subroutine descriptions in the following sections contain additional information on argument usage. The IBUF, BUF, and ICHN (Random Channel List address) arguments must be aligned on specific boundaries. (The VAX-11 FORTRAN User's Guide describes the alignment of FORTRAN arguments.)

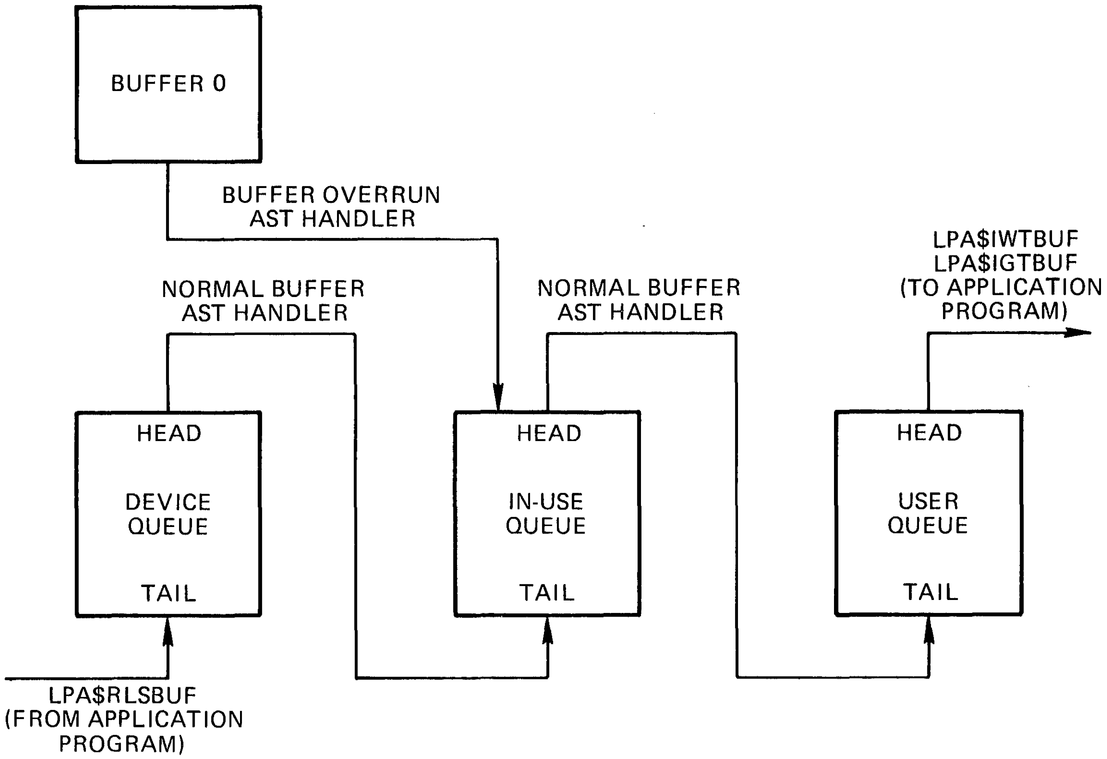


Figure 10-4 Buffer Queue Control

Table 10-5
Subroutine Argument Usage

Argument	Meaning
IBUF	A 50-longword array initialized by the LPA$SETIBF subroutine. IBUF is the impure area used by the buffer management subroutines. A unique IBUF array is required for each simultaneously active request. IBUF must be longword-aligned. The first quadword in the IBUF array is an I/O status block (IOSB) for high-level language subroutines. The LPA$IGTBUF and LPA$IWTBUF subroutines fill this quadword with the current and completion status (see Section 10.6).
LBUF	Specifies the size of each data buffer in words (must be even for dedicated mode sweeps). All buffers are the same size. The minimum value for LBUF is 1 for multirequest mode data transfers and 258 for dedicated mode data transfers. The aggregate size of the assigned buffers must be less than 32,768 words. Thus, the maximum size of each buffer (in words) is limited to 32,768 divided by the number of buffers. The LBUF argument length is one word.

(continued on next page)

Table 10-5 (Cont.)
Subroutine Argument Usage

Argument	Meaning
NBUF	Specifies the number of times the buffers are to be filled during the life of the request. If 0 (default) is specified, sampling is indefinite and must be stopped with the LPA$STPSWP subroutine. The NBUF argument length is one longword.
MODE	Specifies sampling options. MODE bit values are listed in the appropriate subroutine descriptions. The default is 0. MODE values can be added to specify several options. No options are mutually exclusive although not all bits may be applicable at the same time. The MODE argument length is one word.
IRATE	Specifies the clock rate: 0 = Clock B overflow or no rate 1 = 1 MHz 2 = 100 kHz 3 = 10 kHz 4 = 1 kHz 5 = 100 Hz 6 = Schmidt trigger 7 = Line frequency The IRATE argument length is one longword.
IPRSET	Specifies the hardware clock preset value. This value is the 2's complement of the desired number of clock ticks between clock interrupts. (The maximum value is the 2's complement of 65,536.) IPRSET can be computed by the LPA$XRATE subroutine. The IPRSET argument length is one word.
DWELL	Specifies the number of hardware clock overflows between sample sequences in multirequest mode. For example, if DWELL is 20 and NCHN is 3, then after 20 clock overflows one channel is sampled on each of the next three successive overflows; no sampling occurs for the next 20 clock overflows. This allows different users to use different sample rates with the same hardware clock overflow rate. In dedicated mode, the hardware clock overflow rate controls sampling and DWELL is not accessed. Default for DWELL is 1. The DWELL argument length is one word.

(continued on next page)

Table 10-5 (Cont.)
Subroutine Argument Usage

Argument	Meaning
IEFN	Specifies the event flag number or completion routine address. The selected event flag is set at the end of each buffer transaction. If IEFN is 0 (default), event flag 22 is used. IEFN can also specify the address of a completion routine. This routine is called by the buffer management routine when a buffer is available and when the request is terminated, either successfully or with an error. The standard VAX/VMS calling and return sequences are used. The completion routine is called from an AST routine and is therefore at AST level. If IEFN specifies the address of a completion routine, the program must call LPA$IGTBUF to obtain the next buffer. If IEFN specifies an event flag, the program must call LPA$IWTBUF to obtain the next buffer and must use the %VAL operator: ,%VAL(3), (Event flag 3) ,BFRFULL, (Address of completion routine) The IEFN argument length is one longword. If multiple sweeps are initiated, they must use different event flags (the software does not enforce this policy). Event flag 23 is reserved for use by the LPA$CLOCKA and LPA$CLOCKB subroutines. If either of these subroutines is included in the user program, event flag 23 cannot be used. Also, if IEFN is defaulted, event flag 22 cannot be used in the user program.
LDELAY	Specifies the delay, in IRATE units, from the start event until the first sample is taken. The maximum value is 65,536; default is 1. The LDELAY argument length is one word. The LPA11-K supports the LDELAY argument in multirequest mode only.
ICHN	Specifies the number of the first I/O channel to be sampled. Default is channel 0. The ICHN argument length is one byte. The channel number is not the same as the channel assigned to the device by the $ASSIGN system service (see Section 1.8.1). The LPA11-K uses the channel number to specify the multiplexer address of an A/D, D/A, or digital I/O device on the LPA11-K internal I/O bus.

(continued on next page)

Table 10-5 (Cont.)
Subroutine Argument Usage

Argument	Meaning
NCHN	Specifies the number of I/O device channels to sample in a sample sequence. Default is 1. If the NCHN argument is 1, the single channel bit is set in the mode word of the start Request Descriptor Array (RDA) when the sweep is started. The RDA contains the information needed by the LPA11-K for each command (see the LPA11-K Laboratory Peripheral Accelerator User's Guide). The NCHN argument length is one word.
IND	Receives the VAX/VMS success or failure code of the call. The IND argument length is one longword.

10.5.2 LPA$ADSWP - Initiate Synchronous A/D Sampling Sweep

The LPA$ADSWP subroutine initiates A/D sampling through an AD11-K.

The format of the LPA$ADSWP call is as follows:

```
    CALL LPA$ADSWP (IBUF,LBUF,[NBUF],[MODE],[DWELL],[IEFN],[LDELAY],
                   [ICHN],[NCHN],[IND])
```

Arguments are as described in Section 10.5.1.2, with the following additions:

MODE   Specifies sampling options. VAX/VMS defines the following sampling option values:

Value	Meaning
32	Parallel A/D conversion sample algorithm is used if dual A/D converters are specified (value = 8192). Absence of this bit implies the serial A/D conversion sample algorithm.
64	Multirequest mode request. Absence of this bit implies a dedicated mode request.
512	External trigger (Schmidt trigger 1). Dedicated mode only. (The LPA11-K Laboratory Peripheral Accelerator User's Guide describes the use of an external trigger.)
1024	Time stamped sampling with Clock B. The double word consists of one data word followed by the value of the LPA11-K's internal 16-bit counter at the time of the sample (see Section 2.4.3 in the LPA11-K Laboratory Peripheral Accelerator User's Guide). Multirequest mode only.
2048	Event marking. Multirequest mode only. (The LPA11-K Laboratory Peripheral Accelerator User's Guide describes event marking.)

Value	Meaning
4096	Start method. If selected, digital input start. If not selected, immediate start. Multirequest mode only.
8192	Dual A/D converters are to be used. Dedicated mode only.
16384	Buffer overrun is a nonfatal error. The LPA11-K will automatically default to fill buffer 0 if a buffer overrun condition occurs.

If MODE is defaulted, A/D sampling starts immediately with absolute channel addressing in dedicated mode. The LPA11-K does not support delays in dedicated mode.

IND Returns the success or failure status:

0 = Error in call. Possible causes are: LPA$SETIBF was not previously called; LPA$RLSBUF was not previously called; size of data buffers disagrees with the size computed by the LPA$SETIBF call.

1 = successful sweep started

nnn = VAX/VMS status code

10.5.3 LPA$DASWP - Initiate Synchronous D/A Sweep

The LPA$DASWP subroutine initiates D/A output to an AA11-K.

The format for the LPA$DASWP call is as follows:

```
CALL LPA$DASWP (IBUF,LBUF,[NBUF],[MODE],[DWELL],[IEFN],[LDELAY],
          [ICHN],[NCHN],[IND])
```

Arguments are as described in Section 10.5.1.2, with the following additions:

MODE Specifies the sampling options. VAX/VMS defines the following start criteria values:

Value	Meaning
0	Immediate start. This is the default value for MODE.
64	Multirequest mode. If not selected, this request is for dedicated mode.
4096	Start method. If selected, digital input start. If not selected, immediate start. Multirequest mode only.
16384	Buffer overrun is a nonfatal error. The LPA11-K will automatically default to empty buffer 0 if a buffer overrun condition occurs.

IND Returns the success or failure status:

0 = Error in call. Possible causes are: LPA$SETIBF was not previously called; LPA$RLSBUF was not previously called; size of data buffers disagrees with the size computed by the LPA$SETIBF call.

1 = successful sweep started

nnn = VAX/VMS status code

10.5.4 LPA$DISWP - Initiate Synchronous Digital Input Sweep

The LPA$DISWP subroutine initiates digital input through a DR11-K. LPA$DISWP is applicable in multirequest mode only.

The format of the LPA$DISWP call is as follows:

```
CALL LPA$DISWP (IBUF,LBUF,[NBUF],[MODE],[DWELL],[IEFN],[LDELAY],
          [ICHN],[NCHN],[IND])
```

Arguments are as described in Section 10.5.1.2, with the following additions:

MODE Specifies sampling options. VAX/VMS defines the following sampling option values:

Value	**Meaning**
0	Immediate start. This is the default value for MODE.
512	External trigger for DR11-K. (The LPA11-K Laboratory Peripheral Accelerator User's Guide describes the use of an external trigger.)
1024	Time stamped sampling with Clock B. The double word consists of one data word followed by the value of the internal LPA11-K, 16-bit, counter at the time of the sample (see Section 2.4.3 in the LPA11-K Laboratory Peripheral Accelerator User's Guide).
2048	Event marking. (The LPA11-K Laboratory Peripheral Accelerator User's Guide describes event marking.)
4096	Start method. If selected, digital input start. If not selected, immediate start.
16384	Buffer overrun is a non-fatal error. The LPA11-K will automatically default to fill buffer 0 if a buffer overrun condition occurs.

IND Returns the success or failure status:

0 = Error in call. Possible causes are: LPA$SETIBF was not previously called; LPA$RLSBUF was not previously called; size of data buffers disagrees with the size computed by the LPA$SETIBF call.

1 = successful sweep started

nnn = VAX/VMS status code

10.5.5 LPA$DOSWP - Initiate Synchronous Digital Output Sweep

The LPA$DOSWP subroutine initiates digital output through a DR11-K. LPA$DOSWP is applicable in multirequest mode only.

The format of the LPA$DOSWP call is as follows:

```
     CALL LPA$DOSWP (IBUF,LBUF,[NBUF],[MODE],[DWELL],[IEFN],[LDELAY],
               [ICHN],[NCHN],[IND])
```

Arguments are as described in Section 10.5.1.2, with the following additions:

MODE Specifies the sampling options. VAX/VMS defines the following values:

Value	**Meaning**
0	Immediate start. This is the default value for MODE.
512	External trigger for DR11-K (The LPA11-K Laboratory Peripheral Accelerator User's Guide describes the use of an external trigger.)
4096	Start method. If selected, digital input start. If not selected, immediate start.
16384	Buffer overrun is a non-fatal error. The LPA11-K will automatically default to empty buffer 0 if a buffer overrun condition occurs.

IND Returns the success or failure status:

0 = Error in call. Possible causes are: LPA$SETIBF was not previously called; LPA$RLSBUF was not previously called; size of data buffers disagrees with the size computed by the LPA$SETIBF call.

1 = successful sweep started

nnn = VAX/VMS status code

10.5.6 LPA$LAMSKS - Set LPA11-K Masks and NUM Buffer

The LPA$LAMSKS subroutine initializes a user buffer which contains a number to append to the logical name LPA11$, a digital start word mask, an event mark mask, and channel numbers for the two masks.

LPA$LAMSKS must be called:

- By users who intend to use digital input starting or event marking
- By users who do not want to use the default of LAA0 assigned to LPA11$0
- If multiple LPA11-Ks are used

The format of the LPA$LAMSKS call is as follows:

```
     CALL LPA$LAMSKS (LAMSKB,[NUM],[IUNIT],[IDSC],[IEMC],[IDSW],
               [IEMW],[IND])
```

Argument descriptions are as follows:

LAMSKB   Specifies a 4-word array.

NUM   Specifies the number appended to LPA11$. The sweep is started on the LPA11-K assigned to LPA11$num.

IUNIT   Not used. This argument is present for compatibility only.

IDSC   Specifies the digital START word channel. Range is 0 through 4. The IDSC argument length is one byte.

IEMC   Specifies the event MARK word channel. Range is 0 through 4. The IEMC argument length is one byte.

IDSW   Specifies the digital START word mask. The IDSW argument length is one word.

IEMW   Specifies the event MARK word mask. The IEMW argument length is one word.

IND   Always equal to 1 (success). This argument is present for compatibility only.

10.5.7 LPA$SETADC - Set Channel Information For Sweeps

The LPA$SETADC subroutine establishes channel start and increment information for the sweep control subroutines (see Table 10-4). The LPA$SETIBF subroutine must be called to initialize IBUF before LPA$SETADC is called.

The two formats for the LPA$SETADC call are as follows:

```
     CALL LPA$SETADC (IBUF,[IFLAG],[ICHN],[NCHN],[INC],[IND])
```

or

```
     IND=LPA$SETADC (IBUF,[IFLAG],[ICHN],[NCHN],[INC])
```

Argument descriptions are as follows:

IND — Returns the success or failure status:

0 = LPA$SETIBF was not called prior to the LPA$SETADC call

1 = LPA$SETADC call successful

IBUF — The IBUF array specified in the LPA$SETIBF call

IFLAG — Reserved. This argument is present for compatibility only.

ICHN — Specifies the first channel number. Range is 0 through 255; default is 0. The ICHN argument length is one longword.

If INC = 0, ICHN is the address of a Random Channel List. This address must be word-aligned.

NCHN — Specifies the number of samples taken per sample sequence. Default is 1.

INC — Specifies the channel increment. Default is 1. If INC is 0, ICHN is the address of a Random Channel List. The INC argument length is one longword.

10.5.8 LPA$SETIBF - Set IBUF Array For Sweeps

The LPA$SETIBF subroutine initializes the IBUF array for use with the LPA$ADSWP, LPA$DISWP, LPA$DOSWP, LPA$DASWP, LPA$STPSWP, LPA$IWTBUF, LPA$IGTBUF, LPA$IBFSTS, LPA$RLSBUF, LPA$INXTBF, LPA$SETADC, and LPA$RMVBUF subroutines.

The format of the LPA$SETIBF call is as follows:

```
     CALL LPA$SETIBF (IBUF,[IND],[LAMSKB],BUF0,[BUF1,...,BUF7])
```

Arguments are as described in Section 10.5.1.2, with the following additions:

IBUF — Specifies a 50-longword array that is initialized by this subroutine. IBUF must be longword-aligned. (See Table 10-5 for additional information on IBUF.)

IND — Returns the success or failure status:

0 = Error in call. Possible causes are: incorrect number of arguments; IBUF array not longword-aligned; buffer addresses not equidistant.

1 = IBUF initialized successfully

LAMSKB — Specifies the name of a 4-word array. This array allows the use of multiple LPA11-Ks within the same program because the argument used to start the sweep is specified by the LPA$LAMSKS call. (See Section 10.5.6 for a description of the LPA$LAMSKS subroutine.)

BUF0, etc. Specify the names of the buffers. A maximum of eight buffers can be specified. At least two buffers must be specified to provide continuous sampling. The LPA11-K driver requires that all buffers be contiguous. To ensure this, LPA$SETIBF verifies that all buffer addresses are equidistant. Buffers must be longword-aligned.

10.5.9 **LPA$STPSWP - Stop In-progress Sweep**

The LPA$STPSWP subroutine allows a user to stop a sweep that is in progress.

The format of the LPA$STPSWP call is as follows:

```
     CALL LPA$STPSWP (IBUF,[IWHEN],[IND])
```

Arguments are as described in Section 10.5.1.2, with the following additions:

IBUF THE IBUF array specified in the LPA$ADSWP, LPA$DASWP, LPA$DISWP, or LPA$DOSWP call that initiated the sweep.

IWHEN Specifies when to stop the sweep. VAX/VMS defines the following values:

0 = Abort sweep immediately. Uses the $CANCEL system service. This is the default sweep stop.

1 = Stop sweep when the current buffer transaction is completed. (This is the preferred way to stop requests.)

IND Receives a success or failure code in the standard VAX/VMS format:

1 = Success

nnn = VAX/VMS error code issued by the $CANCEL system service

10.5.10 **LPA$CLOCKA - Clock A Control**

The LPA$CLOCKA subroutine sets the clock rate for Clock A.

The format of the LPA$CLOCKA call is as follows:

```
     CALL LPA$CLOCKA (IRATE,IPRSET,[IND],[NUM])
```

Arguments are as described in Section 10.5.1.2, with the following additions:

IRATE — Specifies the clock rate. One of the following values must be specified:

0 = Clock B overflow or no rate
1 = 1 MHz
2 = 100 kHz
3 = 10 kHZ
4 = 1 kHz
5 = 100 Hz
6 = Schmidt trigger 1
7 = Line frequency

IPRSET — Specifies the clock preset value. Maximum of 16 bits. The LPA$XRATE subroutine can be used to calculate this value. The clock rate divided by the clock preset value yields the clock overflow rate.

IND — Receives a success or failure code as follows:

1 = Clock A set successfully

nnn = VAX/VMS error code indicating an I/O error

NUM — Specifies the number to be appended to the logical name LPA11$. If defaulted, NUM is 0. This subroutine sets Clock A on the LPA11-K assigned to LPA11$num.

10.5.11 LPA$CLOCKB - Clock B Control

The LPA$CLOCKB subroutine provides the user with control of the KW11-K Clock B.

The format of the LPA$CLOCKB call is as follows:

```
     CALL LPA$CLOCKB ([IRATE],IPRSET,MODE,[IND],[NUM])
```

Arguments are as described in Section 10.5.1.2, with the following additions:

IRATE — Specifies the clock rate. One of the following values must be specified:

0 = Stops Clock B
1 = 1 MHz
2 = 100 kHz
3 = 10 kHz
4 = 1 kHz
5 = 100 Hz
6 = Schmidt trigger 3
7 = Line frequency

If IRATE is 0 (default), the clock is stopped and the IPRSET and MODE arguments are ignored.

IPRSET — Specifies the preset value by which the clock rate is divided to yield the overflow rate. Maximum of 8 bits. Overflow events can be used to drive Clock A. The LPA$XRATE subroutine can be used to calculate the IPRSET value.

MODE — Specifies options. VAX/VMS defines the following values:

1 = Clock B operates in noninterrupt mode.

2 = The feed B to A bit in the Clock B status register will be set (see Section 3.3 of the *LPA11-K Laboratory Peripheral Accelerator User's Guide*).

IND — Receives a success or failure code as follows:

1 = Clock B set successfully

nnn = VAX/VMS error code indicating an I/O error

NUM — Specifies the number to be appended to the logical name LPA11$. If defaulted, NUM is 0. This subroutine sets Clock B on the LPA11-K assigned to LPA11$num.

10.5.12 LPA$XRATE - Compute Clock Rate and Preset Value

The LPA$XRATE subroutine computes the clock rate and preset value for the LPA$CLOCKA and LPA$CLOCKB subroutines using the specified intersample interval (AINTRVL).

The two formats for the LPA$XRATE call are as follows:

```
CALL LPA$XRATE (AINTRVL,IRATE,IPRSET,IFLAG)
```

or

```
ACTUAL=LPA$XRATE(AINTRVL,IRATE,IPRSET,IFLAG)
```

Arguments are as described in Section 10.5.1.2, with the following additions:

AINTRVL — Specifies the intersample time selected by the user. The time is expressed in decimal seconds. Data type is floating point.

IRATE — Receives the computed clock rate as a value from 1 through 5.

IPRSET — Receives the computed clock preset value.

IFLAG — If the computation is for Clock A, IFLAG is 0; if for Clock B, IFLAG is not 0 (the maximum preset value is 255). The IFLAG argument length is one byte.

ACTUAL — Receives the actual intersample time if called as a function. Data type is floating point. If there are truncation and roundoff errors, this time can be different from the specified intersample time. Note that when LPA$XRATE is called from VAX-11 FORTRAN programs as a function, it must be explicitly declared a real function. Otherwise, LPA$XRATE defaults to an integer function.

If AINTRVL is too large or too small to be achieved, both IRATE and ACTUAL are returned to 0.

10.5.13 LPA$IBFSTS - Return Buffer Status

The LPA$IBFSTS subroutine returns information on the buffers used in a sweep.

The format of the LPA$IBFSTS call is as follows:

```
CALL LPA$IBFSTS (IBUF,ISTAT)
```

Argument descriptions are as follows:

IBUF — The IBUF array specified in the call that initiated the sweep.

ISTAT — Specifies a longword array with as many elements as there are buffers involved in the sweep (maximum of eight). LPA$IBFSTS fills each array element with the status of the corresponding buffer:

+2 = Buffer in device queue. LPA$RLSBUF has been called for this buffer.

+1 = Buffer in user queue. The LPA11-K has filled (data input) or emptied (data output) this buffer.

0 = Buffer is not in any queue.

-1 = Buffer is in the in-use queue, that is, it is either being filled or emptied or is the next to be filled or emptied by the LPA11-K.

10.5.14 LPA$IGTBUF - Return Buffer Number

The LPA$IGTBUF subroutine returns the number of the next buffer to be processed by the application program, that is, the buffer at the head of the user queue (see Figure 10-4). LPA$IGTBUF should be called by a completion routine at AST level to determine the next buffer to process. If an event flag was specified in the start sweep call, LPA$IWTBUF, not LPA$IGTBUF, should be called.

The formats of the LPA$IGTBUF call are as follows:

```
CALL LPA$IGTBUF (IBUF,IBUFNO)
```

or

```
IBUFNO=LPA$IGTBUF(IBUF)
```

Arguments are as described in Section 10.5.1.2, with the following additions:

IBUF — The IBUF array specified in the call that initiated the sweep.

IBUFNO — Returns the number of the next buffer to be filled or emptied by the application program.

Table 10-6 lists the possible combinations of IBUFNO and IOSB contents on the return from a call to LPA$IGTBUF. The first four words of the IBUF array contain the IOSB. If IBUFNO is -1, the IOSB must be checked to determine the reason.

Table 10-6
LPA$IGTBUF Call - IBUFNO and IOSB Contents

IBUFNO	IOSB(1)	IOSB(2)	IOSB(3),(4)	Meaning
n	0	(byte count)	0	Normal buffer complete.
-1	0	0	0	No buffers in queue. Request still active.
-1	1	0	0	No buffers in queue. Sweep terminated normally.
-1	VAX/VMS error code	0	LPA11-K ready-out and maint. registers (only if SS$_DEVREQERR, SS$_CTRLERR, or SS$_DEVCMDERR is returned)	No buffers in queue. Sweep terminated due to error condition. Section 10.6 describes the VAX/VMS error codes; Appendix A of the LPA11-K Laboratory Peripheral Accelerator User's Guide lists the LPA11-K error codes.

10.5.15 LPA$INXTBF - Set Next Buffer to Use

The LPA$INXTBF subroutine alters the normal buffer selection algorithm to allow the user to specify the next buffer to be filled or emptied. The specified buffer is reinserted at the head of the device queue.

The two formats of the LPA$INXTBF call are as follows:

```
CALL LPA$INXTBF (IBUF,IBUFNO,IND)
```

or

```
IND=LPA$INXTBF(IBUF,IBUFNO)
```

Arguments are as described in Section 10.5.1.2, with the following additions:

IBUF — The IBUF array specified in the call that initiated the sweep.

IBUFNO — Specifies the number of the next buffer to be filled or emptied. The buffer must already be in the device queue.

IND — Returns the result of the call:

0 = Specified buffer was not in the device queue

1 = Next buffer was successfully set

10.5.16 LPA$IWTBUF - Return Next Buffer or Wait

The LPA$IWTBUF subroutine returns the next buffer to be processed by the application program, that is, the buffer at the head of the user queue. If the user queue is empty, LPA$IWTBUF waits until a buffer is available. If a completion routine was specified in the call that initiated the sweep, LPA$IGTBUF, not LPA$IWTBUF, should be called.

The two formats of the LPA$IWTBUF call are as follows:

```
CALL LPA$IWTBUF (IBUF,[IEFN],IBUFNO)
```

or

```
IBUFNO=LPA$IWTBUF(IBUF,[IEFN])
```

Arguments are as described in Section 10.5.1.2, with the following additions:

IBUF — The IBUF array specified in the call that initiated the sweep.

IEFN — Not used. This argument is present for compatibility only. (The event flag is the one specified in the start sweep call.)

IBUFNO — Returns the number of the next buffer to be filled or emptied by the application program.

Table 10-7 lists the possible combinations of IBUFNO and IOSB contents on the return from a call to LPA$IWTBUF. The first four words of the IBUF array contain the IOSB. If IBUFNO is -1, the IOSB must be checked to determine the reason.

Table 10-7
LPA$IWTBUF Call - IBUFNO and IOSB Contents

IBUFNO	IOSB(1)	IOSB(2)	IOSB(3),(4)	Meaning
n	0	(byte count)	0	Normal buffer complete.
-1	1	0	0	No buffers in queue. Sweep terminated normally.
-1	VAX/VMS error code	0	LPA11-K ready-out and maint. registers (only if SS$_DEVREQERR, SS$_CTRLERR, or SS$_DEVCMDERR is returned)	No buffers in queue. Sweep terminated due to error condition. Section 10.6 describes the VAX/VMS error codes; Appendix A of the *LPA11-K Laboratory Peripheral Accelerator User's Guide* lists the LPA11-K error codes.

10.5.17 LPA$RLSBUF - Release Data Buffer

The LPA$RLSBUF subroutine declares one or more buffers available to be filled or emptied by the LPA11-K. LPA$RLSBUF inserts the buffer at the tail of the device queue (see Figure 10-4).

The format of the LPA$RLSBUF call is as follows:

```
     CALL LPA$RLSBUF (IBUF,[IND],INDEX0,INDEX1,...,INDEXN)
```

Arguments are as described in Section 10.5.1.2, with the following additions:

IBUF — The IBUF array specified in the call that initiated the sweep.

IND — Returns the success or failure status:

0 = Illegal buffer number or incorrect number of arguments specified, or a double buffer overrun occurred. A double buffer overrun can occur if buffer overrun was specified as a nonfatal error, a buffer overrun occurs, and buffer 0 was not released (probably on the user queue after a previous buffer overrun). LPA$RLSBUF can return a double buffer overrun error only if buffer overrun was specified as a nonfatal error.

1 = Buffer(s) released successfully

INDEX0, etc. — Specify the indexes (0-7) of the buffers to be released. A maximum of eight indexes can be specified.

The LPA$RLSBUF subroutine must be called to release a buffer (or buffers) to the device queue before the sweep is initiated. (See Section 10.5.1.1 for a discussion on buffer management.) Note that LPA$RLSBUF does not verify whether or not the specified buffers are already in a queue. If a buffer is released when it is already in a queue, the queue pointers will be invalidated. This can cause unpredictable results.

If buffer overrun is specified as a nonfatal error, buffer 0 should not be released before the sweep is initiated. However, if either LPA$IGTBUF or LPA$IWTBUF returns buffer 0, it should be released. Note that, in this case, buffer 0 is set aside (not placed on a queue) until the buffer overrun occurs. If a buffer overrun occurs and buffer 0 was not released, the LPA$RLSBUF routine returns an error the next time buffer 0 is released.

10.5.18 LPA$RMVBUF - Remove Buffer from Device Queue

The LPA$RMVBUF subroutine removes a buffer from the device queue.

The format of the LPA$RMVBUF call is as follows:

```
     CALL LPA$RMVBUF (IBUF,IBUFNO,[IND])
```

Arguments are as described in Section 10.5.1.2, with the following additions:

IBUF — The IBUF array specified in the call that initiated the sweep.

IBUFNO — Specifies the number of the buffer to remove from the device queue.

IND — Returns the success or failure status:

0 = Buffer not found in the device queue

1 = Buffer successfully removed from the device queue

10.5.19 LPA$CVADF - Convert A/D Input to Floating Point

The LPA$CVADF subroutine converts A/D input values to floating point numbers. LPA$CVADF is provided for compatibility reasons.

The formats of the LPA$CVADF call are as follows:

```
     CALL LPA$CVADF (IVAL,VAL)
```

or

```
     VAL=LPA$CVADF(IVAL)
```

Argument descriptions are as follows:

IVAL — Contains the value (bits 11:0) read from the A/D input. Bits 15:12 are 0.

VAL — Receives the floating point value.

10.5.20 LPA$FLT16 - Convert Unsigned 16-bit Integer to Floating Point

The LPA$FLP16 subroutine converts unsigned 16-bit integers to floating point. LPA$FLT16 is provided for compatibility reasons.

The formats of the LPA$FLT16 call are as follows:

```
     CALL LPA$FLT16 (IVAL,VAL)
```

or

```
     VAL=LPA$FLT16(IVAL)
```

Argument descriptions are as follows:

IVAL — An unsigned 16-bit integer.

VAL — Receives the converted value.

10.5.21 LPA$LOADMC - Load Microcode and Initialize LPA11-K

The LPA$LOADMC subroutine provides a program interface to the LPA11-K microcode loader. LPA$LOADMC sends a load request through a mailbox to the loader process to load microcode and initialize an LPA11-K (Section 10.7.1 describes the microcode loader process).

The format of the LPA$LOADMC call is as follows:

```
CALL LPA$LOADMC ([ITYPE][,NUM][,IND][,IERROR])
```

Argument descriptions are as follows:

ITYPE The type of microcode to be loaded. VAX/VMS defines the following values:

Value	Meaning
1	Multirequest mode
2	Dedicated A/D mode
3	Dedicated D/A mode

If the ITYPE argument is defaulted, multirequest mode microcode is loaded.

NUM The number to be appended to the logical name LPA11$. If defaulted, NUM is 0.

IND Receives the completion status:

1 = Microcode loaded successfully.

nnn = VAX/VMS error code

IERROR Provides additional error information. Receives the second longword of the IOSB if either SS$_CTRLERR, SS$_DEVCMDERR, or SS$_DEVREQERR is returned in IND. Otherwise, the contents of IERROR is undefined.

10.6 I/O STATUS BLOCK

The I/O status block format for the load microcode, start microprocessor, initialize LPA11-K, set clock, and start data transfer request QIO functions is shown in Figure 10-5.

31 16	15 0
byte count	status
LPA11-K maintenance status	LPA11-K ready-out

Figure 10-5 I/O Functions IOSB Content

VAX/VMS status values and the byte count are returned in the first longword. Status values are defined by the $SSDEF macro. The byte count is the number of bytes transferred by a IO$_LOADMCODE request. If SS$_CTRLERR, SS$_DEVCMDERR, or SS$_DEVREQERR is returned in the status word, the second longword contains the LPA11-K Ready-Out Register and LPA11-K Maintenance Status Register values present at the completion of the request. The high byte of the Ready-Out Register contains the specific LPA11-K error code (see Appendix A of the LPA11-K Laboratory Peripheral Accelerator User's Guide). Table 10-8 lists the status returns for LPA11-K I/O functions.

If high-level language library procedures are used, the status returns listed in Table 10-8 can be returned from the resultant QIO functions. Since buffers are filled by these procedures asynchronously, two I/O status blocks are provided in the IBUF array: one for the high-level language procedures and one for the LPA11-K driver. The first four words of the IBUF array contain the IOSB for the high-level language procedures.

Table 10-8
LPA11-K Status Returns for I/O Functions

Status	Meaning
SS$_ABORT	Request aborted. A request in progress was cancelled by the $CANCEL system service. (Only for start data transfer request functions.)
SS$_BUFNOTALIGN	Alignment error. If this error occurs for an initialize LPA11-K request, the initialize command table was not word-aligned. If this error occurs for a start data transfer request, there are several possible causes: ● User status word (USW) not word-aligned ● Buffer area not longword-aligned ● Random Channel List (RCL) not word-aligned
SS$_CANCEL	Request cancelled by the $CANCEL system service before it started. (Only for the initialize LPA11-K, set clock, and start data transfer request functions.)
SS$_CTRLERR	Controller error. (Only for the start microprocessor, initialize LPA11-K, set clock, and start data transfer request functions.) This is a fatal error that affects all LPA11-K activity. If this error occurs, the LPA11-K terminates all active requests. The third and fourth words of the IOSB contain the LPA11-K Ready-out Status Register and Maintenance Register contents. In particular, the high byte of the third word contains the specific LPA11-K error code (see Appendix A in the LPA11-K Laboratory Peripheral Accelerator User's Guide).

(continued on next page)

Table 10-8 (Cont.)
LPA11-K Status Returns for I/O Functions

Status	Meaning
SS$_DATACHECK	Data check error. (Only for the load microcode function.) A mismatch between the microcode in memory and the microcode loaded into the LPA11-K was detected. The second word of the IOSB contains the number of bytes successfully loaded.
SS$_DEVACTIVE	Device is active. (Only for the load microcode and start microprocessor functions.) The microcode cannot be loaded or the microprocessor cannot be started because there is an active data transfer request.
SS$_DEVCMDERR	LPA11-K command error. (Only for the initialize LPA11-K, set clock, and start data transfer request functions.) This error is associated with the issuance of a new LPA11-K command. The third and fourth words of the IOSB contain the LPA11-K Ready-Out Status Register and Maintenance Register contents. In particular, the high byte of the third word contains the specific LPA11-K error code. (See Appendix A in the LPA11-K Laboratory Peripheral Accelerator User's Guide).
SS$_DEVREQERR	LPA11-K user request error. (Only for start data transfer requests.) The third and fourth words of the IOSB contain the LPA11-K Ready-Out Status Register and Maintenance Register contents. In particular, the high byte of the third word contains the specific LPA11-K error code. (See Appendix A in the LPA11-K Laboratory Peripheral Accelerator User's Guide).
SS$_EXQUOTA	AST quota exceeded. (Only for start data transfer requests.) An AST cannot be queued for a buffer full/empty AST. Normally, a start data transfer request can require no more than three AST blocks at a time.
SS$_INSFBUFDP	A UBA-buffered datapath was not available for allocation. (Only for start data transfer requests in dedicated mode.)
SS$_INSFMAPREG	Insufficient UBA map registers to map the command table or buffer areas. (Only for the initialize LPA11-K and start data transfer request functions.) If the map registers were preallocated when the driver was loaded, the preallocation should be increased.
SS$_INSFMEM	Insufficient dynamic memory to start request or allocate an AST block. (Only for start data transfer requests.)

(continued on next page)

Table 10-8 (Cont.)
LPA11-K Status Returns for I/O Functions

Status	Meaning
SS$_IVBUFLEN	Incorrect length. If this error occurs for an initialize LPA11-K request, the initialize command table length is not the required 278 bytes. If this error occurs for a start data transfer request, there are several possible causes: ● Command table length is not the required 40 bytes ● Buffer area size is not evenly divisible by the number of buffers assigned ● Individual buffer size is 0 ● Individual buffer size is not a multiple of 2 for a multirequest mode request, or 4 for a dedicated mode request ● Random Channel List length is 0 or not a multiple of 2 ● Bit 15 in the last word of the Random Channel List is not set
SS$_IVMODE	Invalid mode. (Only for the initialize LPA11-K function.) The first three bits (2:0) of the first word in the command table, that is, the mode word, are not 0.
SS$_MCNOTVALID	Microcode has not been successfully loaded. (Only for the start microprocessor, initialize LPA11-K, set clock, and start data transfer request functions.)
SS$_PARITY	Parity error. (Only for start data transfer request in deicated mode.) A parity error occurred in a UBA-buffered datapath.
SS$_POWERFAIL	A power failure occurred while a request was active. (Only for the start microprocessor, initialize LPA11-K, set clock, and start data transfer request functions.)
SS$_TIMEOUT	Device timeout. (Only for the start microprocessor, initialize LPA11-K, set clock, and start data transfer request functions.) An interrupt was not received within one second after the request was issued.

10.7 LOADING LPA11-K MICROCODE

The microcode loading and device initialization routines automatically load microcode on system initialization (if specified in the system manager's startup file) and on power recovery. These routines also allow a nonprivileged user to load microcode and restart the system.

The LPA11-K loader and initialization routines consist of three parts:

- A microcode loader process which loads any of the three microcode versions, initializes the LPA11-K, and sets the clock rate. Loading is initiated by either a mailbox request or a power recovery AST. This process requires permanent mailbox (PRMMBX) and physical I/O privileges.

- An operator process which accepts operator commands or indirect file commands to load microcode and initialize an LPA11-K. This process uses a mailbox to send a load request to the loader process; temporary mailbox (TMPMBX) privilege is required.

- An LPA11-K procedure library routine that provides a program interface to the LPA11-K microcode loader. The procedure sends a load request through a mailbox to the loader process to load microcode and initialize an LPA11-K. Section 10.5.21 describes this routine in greater detail.

10.7.1 Microcode Loader Process

The microcode loader process loads microcode, initializes a specific LPA11-K, and sets the clock at the default rate (10 kHz interrupt rate). A bit set in a controller bitmap indicates that the specified controller was loaded. The process specifies a power recovery AST, creates a mailbox whose name (LPA$LOADER) is entered in the system logical name table, and then hibernates.

The correct device configuration is determined automatically. When LPA11-K initialization is performed, every possible device (see Table 10-1) is specified as present on the LPA11-K. If the LPA11-K returns a device not found error, the LPA11-K is reinitialized with that device omitted.

On receipt of a power recovery AST, the loader process examines the controller bitmap to determine which LPA11-Ks have been loaded. For each LPA11-K, the loader process performs the following functions:

- Obtains device characteristics

- Reloads the microcode previously loaded

- Reinitializes the LPA11-K

- Sets Clock A to the previous rate and preset value

10.7.2 Operator Process

The operator process loads microcode and initializes an LPA11-K through the use of either terminal or indirect file commands. The command input syntax is as follows:

```
devname/type
```

Devname is the device name of the LPA11-K to be loaded. A logical name can be specified. However, only one level of logical name translation is performed. If devname is omitted, LAA0 is the default name. If /type appears, it specifies one of three types of microcode to load:

```
/MULTI_REQUEST = multirequest mode
/ANALOG_DIGITAL = dedicated A/D mode
/DIGITAL_ANALOG = dedicated D/A mode
```

If /type is omitted, /MULTI_REQUEST is the default.

After receiving the command, the operator process formats a message and sends it to the loader process. Completion status is returned through a return mailbox.

10.8 RSX-11M VERSION 3.1 AND VAX/VMS DIFFERENCES

This section lists those areas where the VAX/VMS and RSX-11M Version 3.1 LPA11-K high-level language support routines differ. The RSX-11M I/O Drivers Reference Manual provides a detailed description of the RSX-11M LPA11-K support routines. The exact differences between the VAX/VMS and RSX-11M routines can be determined by comparing the descriptions in the RSX-11M manual with the descriptions for the VAX/VMS routines in the preceding sections of this guide.

10.8.1 Alignment and Length

In VAX/VMS:

- Buffers must be contiguous.
- Buffers must be longword-aligned.
- The Random Channel List must be word-aligned.
- The IBUF array length is 50 longwords and must be longword-aligned.

10.8.2 Status Returns

In VAX/VMS:

- The I/O Status Block length is 8 bytes; numeric values of errors are different.
- Several routines return:

  1 - Success

  0 - Failure detected in support routine

  nnn - VAX/VMS status code. Failure detected in system service.

10.8.3 Sweep Routines

In VAX/VMS:

- If an event flag is specified, it must be within a %VAL( ) construction.
- A tenth argument, IND, has been added to return the success or failure status.

10.8.4 General

In VAX/VMS:

- The LUN argument is not used. Instead, the NUM argument specifies the number to be appended to the logical name LPA11$.
- All routine names have the prefix LPA$.
- In the LPA$SETIBF routine, buffer addresses are checked for contiguity.
- In the LPA$LAMSKS routine, the IUNIT argument is not used.
- In the LPA$IWTBUF routine, the IEFN argument is not used. The event flag specified in the sweep routine is used.
- The combinations of IBUFNO and I/O Status Block values returned by the LPA$IWTBUF and LPA$IGTBUF routines are different.

10.9 PROGRAMMING EXAMPLES

The following program examples use LPA11-K high level language procedures and LPA11-K Queue I/O functions.

Appendix B of the VAX/VMS Real-Time User's Guide contains information on LPA11-K programming and design considerations.

10.9.1 LPA11-K High Level Language Program (Program A)

This program is an example of how the LPA11-K high level language procedures perform an A/D sweep using three buffers. The program uses default arguments whenever possible to illustrate the simplest possible calls. The program assumes that dedicated mode microcode has previously been loaded into the LPA11-K. Table 10-9 lists the variables used in this program.

Table 10-9
Program A Variables

Variable	Description
BUFFER	The data buffer array. BUFFER is a common area to guarantee longword-alignment.
IBUF	The LPA11-K high level language procedures use the IBUF array for local storage.
BUFNUM	BUFNUM contains the buffer number returned by LPA$IWTBUF. In this example, the possible values are 0, 1, and 2.
ISTAT	ISTAT contains the status return from the high level language calls.

```
C    *********************************************************************
C
C                               PROGRAM A
C
C    ********************************************************************


          INTEGER*2         BUFFER(1000,0:2),IOSB(4)
          INTEGER*4         IBUF(50),ISTAT,BUFNUM

          COMMON/AREA1/BUFFER

          EQUIVALENCE       (IOSB(1),IBUF(1))

C
C    SET CLOCK RATE TO 100 KHZ, CLOCK PRESET TO -10
C
          CALL LPA$CLOCKA(2,-10,ISTAT)
          IF (.NOT. ISTAT) GO TO 950
C
C    INITIALIZE IBUF ARRAY FOR SWEEP
C
          CALL LPA$SETIBF(IBUF,ISTAT,,BUFFER(1,0),BUFFER(1,1),BUFFER(1,2))
          IF (.NOT. ISTAT) GO TO 950
C
C    RELEASE ALL THE BUFFERS. NOTE USE OF BUFFER NUMBERS RATHER THAN
C    BUFFER NAMES.
C
          CALL LPA$RLSBUF(IBUF,ISTAT,0,1,2)
          IF (.NOT. ISTAT) GO TO 950
C
C    START A/D SWEEP
C
```

```
        CALL LPA$ADSWP(IBUF,1000,50,,,,,,,ISTAT)
        IF (.NOT. ISTAT) GO TO 950
C
C  GET NEXT BUFFER FILLED WITH DATA.  IF BUFNUM IS NEGATIVE, THERE
C  ARE NO MORE BUFFERS AND THE SWEEP IS STOPPED.
C
100     BUFNUM = LPA$IWTBUF(IBUF)
        IF (BUFNUM .LT. 0) GO TO 800
C
C  PROCESS DATA IN BUFFER(1,BUFNUM) TO BUFFER (1000,BUFNUM)
        .
        .
        .
(Application-dependent code is inserted at this point)
        .
        .
        .
C  RELEASE BUFFER TO BE FILLED AGAIN
C
200     CALL LPA$RLSBUF(IBUF,ISTAT,BUFNUM)
        IF (.NOT. ISTAT) GO TO 950
        GO TO 100
C
C  THERE ARE NO MORE BUFFERS TO PROCESS. CHECK TO ENSURE THAT THE
C  SWEEP ENDED SUCCESSFULLY. IOSB(1) CONTAINS EITHER 1 OR A
C  VAX/VMS STATUS CODE.
C
800     IF (.NOT. IOSB(1)) CALL LIB$STOP(%VAL(IOSB(1)))
        PRINT *,'SUCCESSFUL COMPLETION'
        GO TO 2000

C
C  ERROR RETURN FROM SUBROUTINE. ISTAT CONTAINS EITHER 0 OR
C  VAX/VMS ERROR CODE.
C
950     IF (ISTAT .NE. 0) CALL LIB$STOP(%VAL(ISTAT))
        PRINT   *,'ERROR IN LPA11-K SUBROUTINE CALL'
2000    STOP
        END

C  **********************************************************************
```

10.9.2 LPA11-K High-level Language Program (Program B)

This program is a more complex example of LPA11-K operations performed by the LPA11-K high-level language procedures. The following operations are demonstrated:

- Program-requested loading of LPA11-K microcode
- Setting the clock at a specified rate
- Use of nondefault arguments whenever possible
- An A/D sweep that uses an event flag
- A D/A sweep that uses a completion routine
- Buffer overrun set (buffer overrun is a non-fatal error)
- Random Channel List addressing
- Sequential Channel addressing

Table 10-10 lists the variables used in this program.

Table 10-10
Program B Variables

Variable	Description
AD	An array of buffers for an A/D sweep (8 buffers of 500 words each)
DA	An array of buffers for a D/A sweep (2 buffers of 2000 words each)
IBUFAD	The IBUF array for an A/D sweep
IBUFDA	The IBUF array for a D/A sweep
RCL	The array containing the Random Channel List
ADIOSB	The array that contains the I/O status block for the A/D sweep. Equivalenced to the beginning of IBUFAD.
DAIOSB	The array that contains the I/O status block for the D/A sweep. Equivalenced to the beginning of IBUFDA.
ISTAT	Contains the status return from the high-level language calls

```
C  ********************************************************************
C
C                               PROGRAM B
C
C  ********************************************************************

        EXTERNAL FILLBF
        REAL*4 LPA$XRATE

        INTEGER*2 AD(500,0:7),DA(2000,0:1),RCL(5),MODE,IPRSET
        INTEGER*2 ADIOSB(4),DAIOSB(4)

        INTEGER*4 IBUFAD(50),IBUFDA(50),LAMSKB(2)
        INTEGER*4 ISTAT,IERROR,IRATE,BUFNUM

        REAL*4 PERIOD

        COMMON  /SWEEP/AD,DA,IBUFAD,IBUFDA

        EQUIVALENCE (IBUFAD(1),ADIOSB(1)),(IBUFDA(1),DAIOSB(1))

        PARAMETER MULTI=1, HBIT='8000'X, LSTCHN=HBIT+7
C
C  SET UP RANDOM CHANNEL LIST.  NOTE THAT THE LAST WORD MUST HAVE BIT
C  15 SET.
C
        DATA RCL/2,6,3,4,LSTCHN/
```

```
C  *******************************************************************
C
C  LOAD MULTIREQUEST MODE MICROCODE AND SET THE CLOCK OVERFLOW RATE
C  TO 5 KHZ
C
C  *******************************************************************
C
C  LOAD MICROCODE ON LPA11-K ASSIGNED TO LPA11$3
C
        CALL LPA$LOADMC(MULTI,3,ISTAT,IERROR)
        IF (.NOT. ISTAT) GO TO 5000
C
C  COMPUTE CLOCK RATE AND PRESET.  SET CLOCK 'A' ON LPA11-K
C  ASSIGNED TO LPA11$3.
C
        PERIOD = LPA$XRATE(.0002,IRATE,IPRSET,0)
        IF (PERIOD .EQ. 0.0) GO TO 5500

        CALL LPA$CLOCKA(IRATE,IPRSET,ISTAT,3)
        IF (.NOT. ISTAT) GO TO 5000

C  *******************************************************************
C
C  SET UP FOR A/D SWEEP
C
C  *******************************************************************
C
C  INITIALIZE IBUF ARRAY. NOTE THE USE OF THE LAMSKB ARGUMENT BECAUSE
C  THE LPA11-K ASSIGNED TO LPA11$3 IS USED.
C
        CALL LPA$SETIBF(IBUFAD,ISTAT,LAMSKB,AD(1,0),AD(1,1),AD(1,2),
        1               AD(1,3),AD(1,4),AD(1,5),AD(1,6),AD(1,7))
        IF (.NOT. ISTAT) GO TO 5000

        CALL LPA$LAMSKS(LAMSKB,3)
C
C  SET UP RANDOM CHANNEL LIST SAMPLING (20 SAMPLES IN A SAMPLE
C  SEQUENCE)
C
        CALL LPA$SETADC(IBUFAD,,RCL,20,0,ISTAT)
        IF (.NOT. ISTAT) GO TO 5000
C
C  RELEASE BUFFERS FOR A/D SWEEP.  NOTE THAT BUFFER 0 IS NOT
C  RELEASED BECAUSE BUFFER OVERRUN WILL BE SPECIFIED AS NON-FATAL.
C
        CALL LPA$RLSBUF(IBUFAD,ISTAT,1,2,3,4,5,6,7)
        IF (.NOT. ISTAT) GO TO 5000

C  *******************************************************************
C
C  SET UP FOR D/A SWEEP
C
C  *******************************************************************
C
C  NOTE THAT THE SAME LAMSKB ARRAY CAN BE USED BECAUSE THE LAMSKB
C  CONTENTS APPLY TO BOTH A/D AND D/A SWEEPS
C
        CALL LPA$SETIBF(IBUFDA,ISTAT,LAMSKB,DA(1,0),DA(1,1))
        IF (.NOT. ISTAT) GO TO 5000
C
C  SET UP SAMPLING PARAMETERS AS FOLLOWS:  INITIAL CHANNEL = 1.
C  NUMBER OF CHANNELS SAMPLED EACH SAMPLE SEQUENCE = 2, CHANNEL
C  INCREMENT = 2, THAT IS, SAMPLE CHANNELS 1 AND 3 EACH SAMPLE
C  SEQUENCE.
```

```
C
        CALL LPA$SETADC(IBUFDA,,1,2,2,ISTAT)
        IF (.NOT. ISTAT) GO TO 5000
C
C  FILL BUFFERS WITH DATA FOR OUTPUT TO D/A
C
        .
        .
        .
(Application dependent code is inserted here to fill buffers
DA(1,0) through DA(2000,0) and DA(1,1) through DA(2000,1) with data)
        .
        .
        .
C
C  RELEASE BUFFERS FOR D/A SWEEP
C
        CALL LPA$RLSBUF (IBUFDA,ISTAT,0,1)
        IF (.NOT. ISTAT) GO TO 5000

C  ******************************************************************
C
C  START BOTH SWEEPS
C
C  ******************************************************************
C
C  START A/D SWEEP. MODE BITS SPECIFY BUFFER OVERRUN IS NON-FATAL AND
C  MULTIREQUEST MODE. SWEEP ARGUMENTS SPECIFY 500 SAMPLES/BUFFER,
C  INDEFINITE SAMPLING, DWELL = 10 CLOCK OVERFLOWS, SYNCHRONIZE USING
C  EVENT FLAG 15, AND A DELAY OF 50 CLOCK OVERFLOWS.
C
        MODE = 16384 + 64
        CALL LPA$ADSWP(IBUFAD,500,0,MODE,10,%VAL(15),50,,,ISTAT)
        IF (.NOT. ISTAT) GO TO 5000
C
C  START D/A SWEEP.  MODE SPECIFIES MULTIREQUEST MODE.  OTHER
C  ARGUMENTS SPECIFY 2000 SAMPLES/BUFFER, FILL 15 BUFFERS, DWELL = 25
C  CLOCK OVERFLOWS, SYNCHRONIZE BY CALLING THE COMPLETION ROUTINE
C  'FILLBF', AND DELAY = 10 CLOCK OVERFLOWS.  (SEE THE FILLBF LISTING
C  AFTER THE PROGRAM B LISTING.)
C
        MODE = 64
        CALL LPA$DASWP(IBUFDA,2000,15,MODE,25,FILLBF,10,,,ISTAT)
        IF (.NOT. ISTAT) GO TO 5000

C  ********************************************************************
C
C  WAIT FOR AN A/D BUFFER AND THEN PROCESS THE DATA IT CONTAINS. D/A
C  BUFFERS ARE FILLED ASYNCHRONOUSLY BY THE COMPLETION ROUTINE FILLBF.
C
C  ********************************************************************
C
C  WAIT FOR A BUFFER TO BE FILLED BY A/D.  IF BUFNUM IS LESS THAN
C  ZERO, THE SWEEP HAS STOPPED (EITHER SUCCESSFULLY OR WITH AN ERROR).
C
100     BUFNUM = LPA$IWTBUF(IBUFAD)
        IF (BUFNUM .LT. 0) GO TO 1000
C
C  THERE IS A/D DATA IN AD(1,BUFNUM) THROUGH AD(500,BUFNUM)
C
        .
        .
        .
```

```
(Process the A/D data with the application dependent code inserted
here)
        .
        .
        .
C
C  ASSUME SWEEP SHOULD BE STOPPED WHEN THE LAST SAMPLE IN BUFFER
C  EQUALS 0. NOTE THAT THE SWEEP ACTUALLY STOPS WHEN THE BUFFER
C  CURRENTLY BEING FILLED IS FULL.  ALSO NOTE THAT LPA$IWTBUF
C  CONTINUES TO BE CALLED UNTIL THERE ARE NO MORE BUFFERS TO PROCESS.
C
        IF (AD(500,BUFNUM) .NE. 0) GO TO 200
        CALL LPA$STPSWP(IBUFAD,1,ISTAT)
        IF (.NOT. ISTAT) GO TO 5000

C
C  AFTER THE DATA HAS BEEN PROCESSED, THE BUFFER IS RELEASED TO BE
C  FILLED AGAIN.  THEN THE NEXT BUFFER IS OBTAINED FROM A/D.
C
200     CALL LPA$RLSBUF(IBUFAD,ISTAT,BUFNUM)
        IF (.NOT. ISTAT) GO TO 5000
        GO TO 100

C
C  ENTER HERE WHEN A/D SWEEP HAS ENDED.  CHECK FOR ERROR OR
C  SUCCESSFUL END. (NOTE: ASSUME THAT THE D/A SWEEP HAS ALREADY
C  ENDED - SEE COMPLETION ROUTINE FILLBF)
C
1000    IF(ADIOSB(1)) GO TO 6000
        CALL LIB$STOP(%VAL(ADIOSB(1)))

C
C  ENTER HERE IF THERE WAS AN ERROR RETURNED FROM ONE OF THE
C  LPA11-K HIGH LEVEL LANGUAGE CALLS. ISTAT CONTAINS EITHER 0
C  OR A VAX/VMS STATUS CODE.
C
5000    IF (ISTAT .NE. 0) CALL LIB$STOP (%VAL(ISTAT))
5500    PRINT *,'ERROR IN LPA11-K SUBROUTINE CALL'
        GO TO 7000

6000    PRINT *,'SUCCESSFUL COMPLETION'
7000    STOP
        END

C  *******************************************************************
C
C  SUBROUTINE FILLBF
C
C  *******************************************************************
C
C  THE FILLBF SUBROUTINE IS CALLED WHENEVER THE D/A HAS EMPTIED A
C  BUFFER, AND THAT BUFFER IS AVAILABLE TO BE REFILLED.  THIS
C  SUBROUTINE GETS THE BUFFER, FILLS IT, AND RELEASES IT BACK TO THE
C  LPA11-K.  NOTE THAT THE D/A SWEEP IS STOPPED AUTOMATICALLY AFTER
C  15 BUFFERS HAVE BEEN FILLED.  ALSO NOTE THAT FILLBF IS CALLED BY
C  AN AST HANDLER.  IT IS THEREFORE CALLED ASYNCHRONOUSLY FROM THE
C  MAIN PROGRAM AT AST LEVEL.  CARE SHOULD BE EXERCISED WHEN ACCESSING
C  VARIABLES THAT ARE COMMON TO BOTH LEVELS.
C

        INTEGER*2  AD(500,0:7),DA(2000,0:1),DAIOSB(4)
        INTEGER*4  IBUFAD(50),IBUFDA(50),BUFNUM,ISTAT
        EQUIVALENCE  (IBUFDA(1),DAIOSB(1))
        COMMON  /SWEEP/AD,DA,IBUFAD,IBUFDA
```

```
C
C  GET BUFFER NUMBER OF NEXT BUFFER TO FILL
C
        BUFNUM = LPA$IGTBUF(IBUFDA)
        IF (BUFNUM .LT. 0) GO TO 3000

C
C  FILL BUFFER WITH DATA FOR OUTPUT TO D/A
        .
        .
        .
(Application dependent code is inserted here to fill buffer
DA(1,BUFNUM) through DA(2000,BUFNUM) with data)
        .
        .
        .
C
C  RELEASE BUFFER
C
        CALL LPA$RLSBUF(IBUFDA,ISTAT,BUFNUM)
        GO TO 4000

C
C  CHECK FOR SUCCESSFUL END OF SWEEP
C
3000    IF(DAIOSB(1)) GO TO 4000

C
C  ERROR IN SWEEP
C
        CALL LIB$STOP(%VAL(DAIOSB(1)))

4000    RETURN
        END
C *******************************************************************
```

10.9.3 LPA11-K QIO Functions Program (Program C)

This sample program uses QIO functions to start an A/D data transfer from an LPA11-K. (The program assumes multirequest mode microcode has been loaded.) Sequential channel addressing is used. The data transfer is stopped after 100 buffers have been filled; no action is taken with the data as the buffers are filled. Note that this program starts the data transfer and then waits until the QIO operation completes.

```
; *******************************************************************
;
;                       PROGRAM C
;
; *******************************************************************

        .TITLE  LPA11-K EXAMPLE PROGRAM
        .IDENT  /V01/

        .PSECT  LADATA,LONG


IOSB:   .BLKQ   1                       ; I/O STATUS BLOCK
COUNT:  .LONG   0                       ; COUNT OF BUFFERS FILLED
```

```
CBUFF:                                  ; COMMAND BUFFER FOR START
                                        ; DATA QIO
        .WORD   ^X20A                   ; MODE = SEQUENTIAL CHANNEL
                                        ; ADDRESSING, A/D, MULTI-
                                        ; REQUEST MODE
        .WORD   3                       ; VALID BUFFER MASK (4
                                        ; BUFFERS)
        .LONG   USW                     ; USER STATUS WORD ADDRESS
        .LONG   4000                    ; AGGREGATE BUFFER LENGTH
        .LONG   DATA_BUFFER0            ; ADDRESS OF DATA BUFFERS
        .LONG   0                       ; NO RANDOM CHANNEL LIST
                                        ; LENGTH
        .LONG   0                       ; NO RANDOM CHANNEL LIST
                                        ; ADDRESS
        .WORD   10                      ; DELAY
        .BYTE   0                       ; START CHANNEL
        .BYTE   1                       ; CHANNEL INCREMENT
        .WORD   16                      ; NUMBER OF SAMPLES IN
                                        ; SAMPLE SEQUENCE
        .WORD   1                       ; DWELL
        .BYTE   0                       ; START WORD NUMBER
        .BYTE   0                       ; EVENT MARK WORD
        .WORD   0                       ; START WORD MASK
        .WORD   0                       ; EVENT MARK MASK
        .WORD   0                       ; FILLS OUT COMMAND BUFFER

USW:    .WORD   0                       ; USER STATUS WORD

        .ALIGN  LONG                    ;BUFFERS MUST BE
                                        ; LONGWORD ALIGNED

DATA_BUFFER0: .BLKW     500             ; DATA BUFFERS
DATA_BUFFER1: .BLKW     500
DATA_BUFFER2: .BLKW     500
DATA_BUFFER3: .BLKW     500

DEVNAME: .LONG  4,LANAME

CHANNEL: .BLKW  1                       ; CONTAINS CHANNEL NUMBER

LANAME: .ASCII  /LAA0/

        .PSECT  LACODE,NOWRT

START:  .WORD   0
        $ASSIGN_S DEVNAME,CHANNEL       ; ASSIGN CHANNEL
        BLBS    R0,5$                   ; NO ERROR
        BRW     ERROR                   ; ERROR

5$:                                     ; SET CLOCK OVERFLOW RATE
                                        ; TO 2 KHZ. (1 MHZ RATE
                                        ; DIVIDED BY 500 PRESET)
        $QIOW_S ,CHANNEL,#IO$_SETCLOCK,-
                IOSB,,,,#1,#^X143,#-500
        BLBC    R0,ERROR                ; ERROR
        MOVZWL  IOSB,R0                 ; PICK UP I/O STATUS
        BLBC    R0,ERROR                ; ERROR

                                        ; START DATA TRANSFER
        CLRW    USW                     ; CLEAR USW (START WITH
                                        ; BUFFER 0)
        MOVL    #100,COUNT              ; FILL 100 BUFFERS
        $QIOW_S ,CHANNEL,#IO$_STARTDATA,-
```

```
                IOSB,,,CBUFF,#40,#BFRAST
        BLBC    R0,ERROR                ; ERROR

; NOTE THAT THE QIO WAITS UNTIL IT FINISHES. NORMALLY, THE DATA IS
; PROCESSED HERE AS THE BUFFERS ARE FILLED. CHECK FOR ERROR WHEN
; THE QIO COMPLETES.

        MOVZWL  IOSB,R0                 ; PICK UP I/O STATUS
        BLBC    R0,ERROR                ; ERROR
        RET                             ; ALL DONE - EXIT

ERROR:                                  ; ENTER HERE IF ERROR.
                                        ; STATUS IN R0.
        PUSHL   R0                      ; PUSH ONTO STACK
        CALLS   #1,LIB$STOP             ; SIGNAL ERROR


BFRAST:                                 ; BUFFER AST ROUTINE.
                                        ; BFRAST IS CALLED WHENEVER
                                        ; A BUFFER IS FILLED.
        .WORD   0
        INCB    USW+1                   ; ADD 1 TO BUFFER NUMBER
        CMPZV   #0,#3,USW+1,#3          ; HANDLE WRAPAROUND

        BLEQ    10$
        CLRB    USW+1                   ; USE BUFFER 0

10$:    DECL    COUNT                   ; DECREMENT BUFFER COUNT
        BGTR    20$
        BISB    #^X40,USW+1             ; ENOUGH BUFFERS FILLED -
                                        ; SET STOP BIT
20$:    BICB    #^X80,USW+1             ; CLEAR DONE BIT
        RET

        .END    START

; ********************************************************************
```

CHAPTER 11

DR32 INTERFACE DRIVER

11.1 SUPPORTED DEVICE

The DR32 is an interface adapter that connects the internal memory bus of a VAX-11 processor to a user-accessible bus called the DR32 Device Interconnect (DDI). Two DR32s can be connected to form a VAX-11 processor-to-processor link. Figure 11-1 shows the relationship of the DR32 to the VAX 11/780 and the DDI.

As a general purpose data port, the DR32 is capable of moving continuous streams of data to or from memory at high speed. Data from a user device to disk storage must go through an intermediate buffer in main memory.

11.1.1 DR32 Device Interconnect

The DR32 Device Interconnect (DDI) is a bidirectional path for the transfer of data and control signals. Control signals sent over the DDI are asynchronous and interlocked; data transfers are synchronized with clock signals. Any connection to the DDI is called a DR-device. The DDI provides a point-to-point connection between two DR-devices, one of which must be a VAX-11 processor. The DR-device connected to the external end of the DDI is called the far end DR-device.

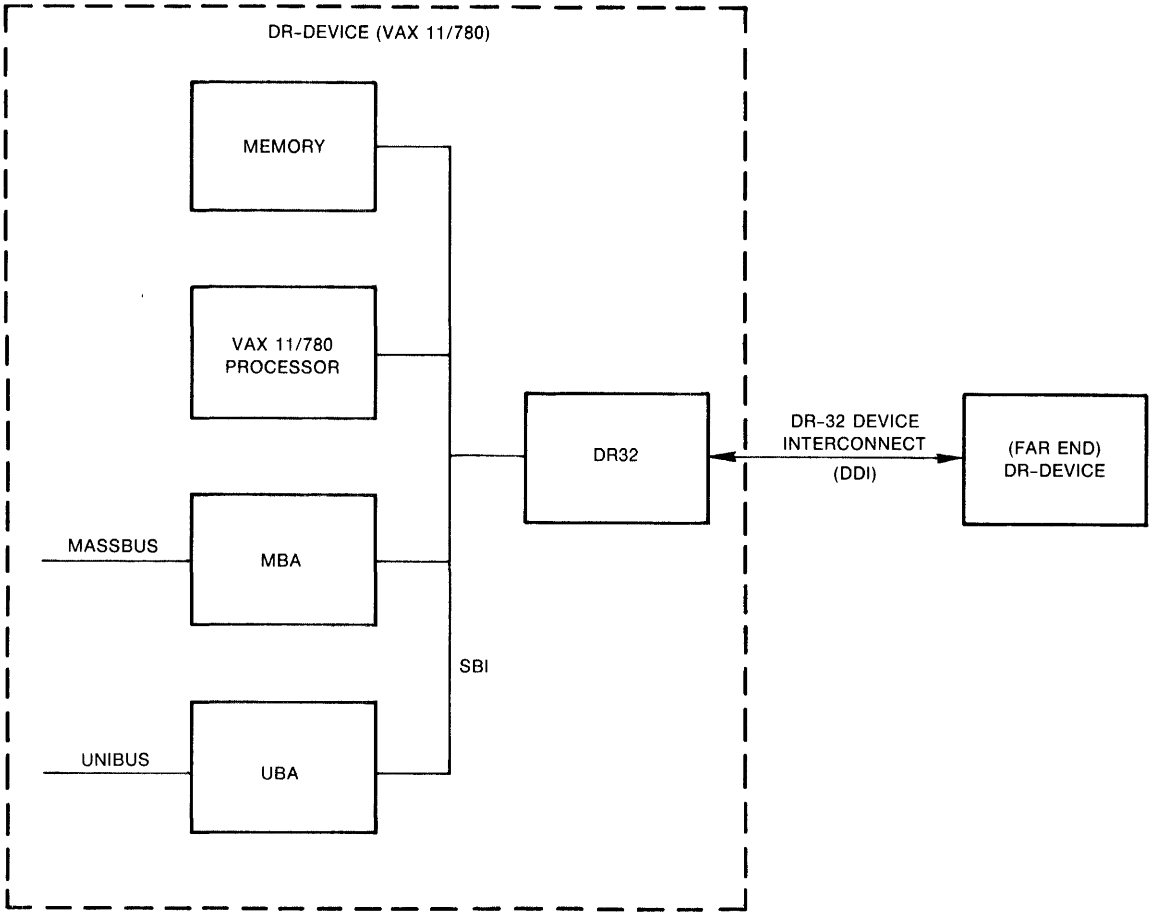


Figure 11-1 Basic DR32 Configuration

11.2 DR32 FEATURES AND CAPABILITIES

The DR32 provides the following features and capabilities:

- 32-bit parallel data transfers
- High bandwidth (6 megabytes/second on the DDI with a VAX 11/780)
- Word or byte alignment of data
- Half-duplex operation
- Hardware-supported (I/O driver-independent) memory mapping
- Separate Control and Data Interconnects
- Command and data chaining
- Direct software link between the DR32 and the user process
- Synchronization of the user program with DR32 data transfers
- Transfers initiated by an external device

The following sections describe the capabilities.

11.2.1 Command and Data Chaining

Command chaining is the execution of commands without software intervention for each command. Commands are chained in the sense that they follow each other on a queue. After a QIO function starts the DR32, any number of DR32 commands can be executed during that QIO operation. This process continues until the transfer is halted (a command packet is fetched that specifies a halt command) or an error occurs.

Command packets can specify data chaining. In data chaining, a number of main memory buffers appear as one large buffer to the far end DR-device. Data chaining is completely transparent to this device; transfers are seen as a continuous stream of data. Chained buffers can be of arbitrary byte alignment and length. The length of a transfer appears to the far end DR-device to be the total of all the byte counts in the chain, and since chains in the DR32 can be of unlimited length, the device sees the byte count as potentially infinite.

11.2.2 Far End DR-device Initiated Transfers

The DR32 provides the capability for the far end DR-device to initiate data transfers to the VAX-11 memory, that is, it provides for random access mode. Random access consists of data transfers to or from the VAX-11 memory without notification of the VAX-11 processor. This mode is used when two DR32s are connected to form a processor-to-processor link. You can discontinue random access by specifying a command packet with random access disabled. It can also be discontinued by an abort from either the controlling process or the far end DR-device.

11.2.3 Power Failure

If power fails on the DR32 but not on the system, the DR32 driver aborts the active data transfer and returns the status code SS$_POWERFAIL in the I/O status block. If a system power-failure occurs, the DR32 driver completes the active data transfer when power is recovered and returns the status code SS$_POWERFAIL.

11.2.4 Interrupts

The DR32 can interrupt the DR32 driver for any of the following reasons:

- An abort has occurred. The QIO is completed.
- A DR32 power-down or power-up sequence has occurred
- An unsolicited control message has been sent to the DR32. If this command packet's interrupt control field is properly set up, a packet AST interrupt occurs. The interrupt occurs after the command packet obtained from FREEQ is placed on TERMQ.
- The DR32 enters the halt state. The QIO is completed.

- A command packet that specifies an unconditional interrupt has been placed onto TERMQ. The result is a packet AST.
- A command packet with the "interrupt when TERMQ empty" bit set was placed on an empty TERMQ. The result is a packet AST.

11.3 DEVICE INFORMATION

Users can obtain information on the DR32 by using the $GETCHN and $GETDEV system services (see Section 1.10). The DR32-specific information is returned in the first three longwords of a user-specified buffer, as shown in Figure 11-2 (Figure 1-9 shows the entire buffer).

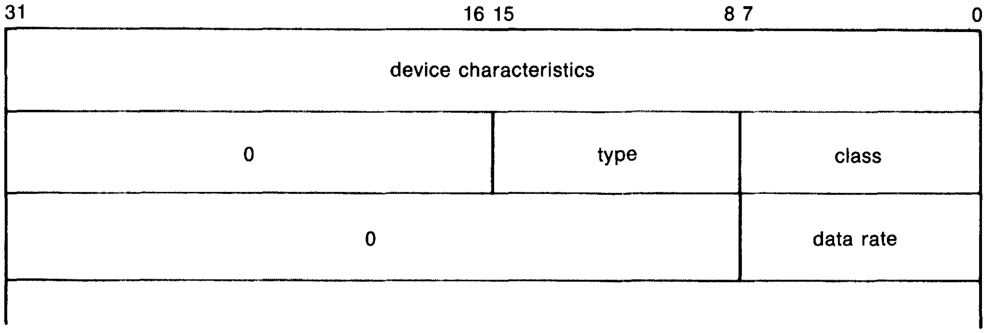


Figure 11-2 DR32 Information

The first longword contains device-independent information. The second and third longwords contain device-dependent data.

Table 11-1 lists the device-independent characteristics returned in the first longword.

Table 11-1
Device-Independent Characteristics

Dynamic Bit[1] (Conditionally Set)	Meaning
DEV$M_AVL	Device is available
Static Bits[1] (Always Set)	
DEV$M_IDV	Input device
DEV$M_ODV	Output device
DEV$M_RTM	Real time device

[1] Defined by the $DEVDEF macro.

The second longword contains information on the device class and type. The device class for the DR32 is DC$_REALTIME and the device type for the DR780 is DT$_DR780. The $XFDEF macro defines these values.

The low order byte of the third longword contains the last data rate value loaded into the DR32 data rate register.

11.4 PROGRAMMING INTERFACE

The DR32 is supported by a device driver, a high-level language procedure library of support routines, and a program for microcode loading.

After issuing a IO$_STARTDATA QIO to the DR32 driver, application programs communicate directly with the DR32 by inserting command packets onto queues. This direct link between the application program and the DR32 provides faster communication by avoiding the necessity of going through the I/O driver.

Two interfaces are provided for accessing the DR32: a QIO interface and a support routine interface. The QIO interface requires that the application program build command packets and insert them onto the DR32 queues. The support routine interface, on the other hand, provides procedures for these functions and, in addition, performs housekeeping functions, such as maintaining command memory.

The support routine interface was designed to be called from high-level languages, such as FORTRAN, where the data manipulation required by the QIO interface might be awkward. Note, however, that the user of the support routines must be equally as sophisticated as the user of the QIO interface in terms of knowledge of the DR32 and the meaning of the fields in the command packets.

11.4.1 DR32 - Application Program Interface

The application program interfaces with the DR32 through two memory areas. These areas are called the command block and the buffer block. The addresses and sizes of the blocks are determined by the application program and passed to the DR32 driver as arguments to the IO$_STARTDATA function. This QIO function starts the DR32 (see Section 11.4.5.2). Both blocks are locked into memory while the DR32 is active. The buffer block defines the area of memory that is accessible to the DR32 for the transfer of data between the far end DR-device and the DR32. The command block contains the headers for the three queues that provide the communication path between the DR32 and the application program, and space in which to build command packets.

The interface between the DR32 and the application program contains three queues: the input queue (INPTQ), the termination queue (TERMQ), and the free queue (FREEQ). Information is transferred between the DR32 and the far end DR-device through the use of command packets. The three queue structures control the flow of command packets to and from the DR32. The application program builds a command packet and inserts it onto INPTQ. The DR32 removes the packet, executes the specified command, enters some status information, and then inserts the packet onto TERMQ. Unsolicited input from the far end DR-device is placed in packets removed from FREEQ and inserted onto TERMQ.

The INPTQ, TERMQ, and FREEQ headers are located in the first six longwords of the command block. Since the queues are self-relative, that is, they use the VAX-11 self-relative queue instructions, the headers must be quadword aligned. The application program must initialize all queue headers. Figure 11-3 shows the position of the queue headers in the command block. Section 11.4.2 describes queue processing in greater detail.

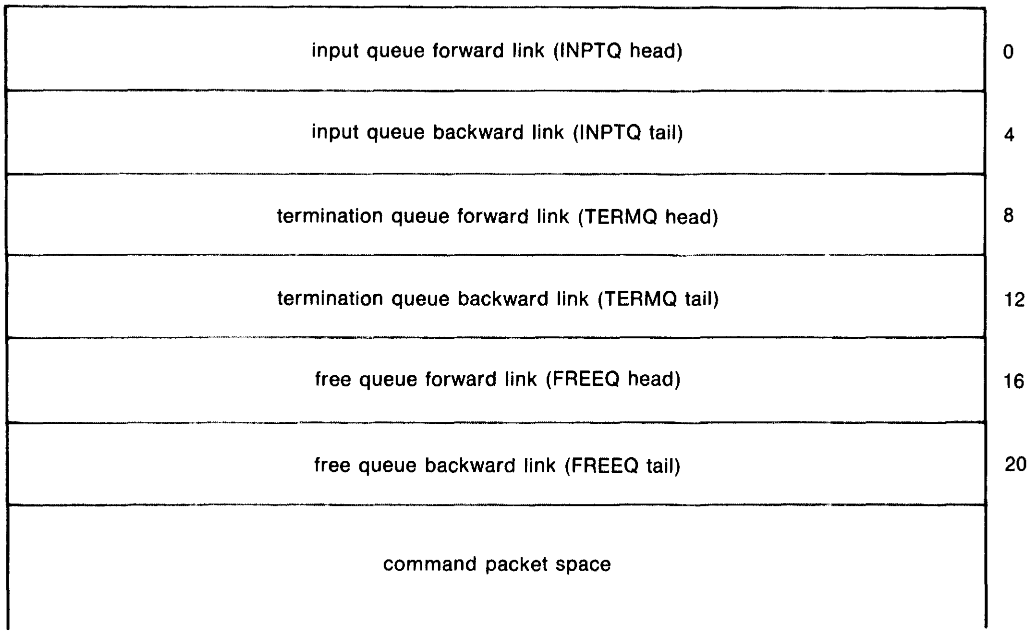

input queue forward link (INPTQ head)
0
input queue backward link (INPTQ tail)
4
termination queue forward link (TERMQ head)
8
termination queue backward link (TERMQ tail)
12
free queue forward link (FREEQ head)
16
free queue backward link (FREEQ tail)
20
command packet space

Figure 11-3 Command Block (Queue Headers)

11.4.2 Queue Processing

Three queue structures control the flow of command packets to and from the DR32:

- Input queue (INPTQ)
- Termination queue (TERMQ)
- Free queue (FREEQ)

The DR32 removes command packets from the heads of FREEQ and INPTQ and inserts command packets onto the tail of TERMQ. For command sequences initiated by the application program, the DR32 removes command packets from the head of INPTQ, processes them, and returns them to the tail of TERMQ. Queue processing is performed by the DR32 with the equivalent of the INSQTI and REMQHI instructions. To remove a packet from INPTQ, the DR32 executes the equivalent of REMQHI HDR, CMDPTR where CMDPTR is a DR32 register used as a pointer to the current command packet and HDR specifies the INPTQ header. To insert a packet onto TERMQ, the DR32 executes the equivalent of INSQTI CMDPTR, HDR. The user process performs similar operations with the queues, inserting packets onto the head or tail of INPTQ and normally removing packets from the head of TERMQ.

If any of the queues are currently being accessed by the DR32, the program's interlocked queue instructions will fail for one of the following reasons:

1. The DR32 is currently removing a packet from INPTQ or FREEQ, or inserting a packet onto TERMQ, and the operation will be completed shortly.

2. The DR32 detects an error condition, for example, an unaligned queue, that prevents it from completing the queue operation. In this case, the transfer is aborted and the I/O status block contains the error that caused the abort.

To distinguish between these two conditions, the application program must include a queue retry mechanism that retries the queue operation a reasonable number of times, for example 25, before determining that an error condition exists. An example of a queue retry mechanism is shown in the program example (see Section 11.7).

If the DR32 discerns that any of the queues are interlocked, it retries the operation until it completes or the DR32 is aborted.

11.4.2.1 **Initiating Command Sequences** - If a command packet is inserted onto an empty INPTQ, the application program must notify the DR32 of this event. This is accomplished by setting bit 0 in a DR32 register, the GO bit. The IO$_STARTDATA QIO returns the GO bit's address to the application program. After notification by the GO bit that there are command packets on its INPTQ, the DR32 continues to process the packets until INPTQ is empty.

The GO bit can be safely set at any time. While processing command packets, the DR32 ignores the GO bit. If the GO bit is set when the DR32 is idle, the DR32 will attempt to remove a command packet from INPTQ. If INPTQ is empty at this time, the DR32 clears the GO bit and returns to the idle state.

11.4.2.2 **Device-Initiated Command Sequences** - If the DR-device that interfaces the far end of the DDI is capable of transmitting unsolicited control messages, messages of this type can be transmitted to the local DR32. These messages are not synchronized to the application program command flow. Therefore, the DR32 uses a third queue, FREEQ, to handle unsolicited messages. Normally, the application program inserts a number of empty command packets onto FREEQ to allow the external device to transmit control messages.

If a control message is received from the far end DR-device, the DR32 removes an empty command packet from the head of FREEQ, fills the device message field of this packet with the control message and, when the transmission is completed, inserts the packet onto the tail of TERMQ. (The device message field in this command packet must be large enough for the entire message or a length error will occur.) The application program then removes the packet from TERMQ. If the command packet is from FREEQ, the XF$M_PKT_FREQPK bit in the DR32 Status Longword is set.

Figure 11-4 shows the DR32 queue flow.

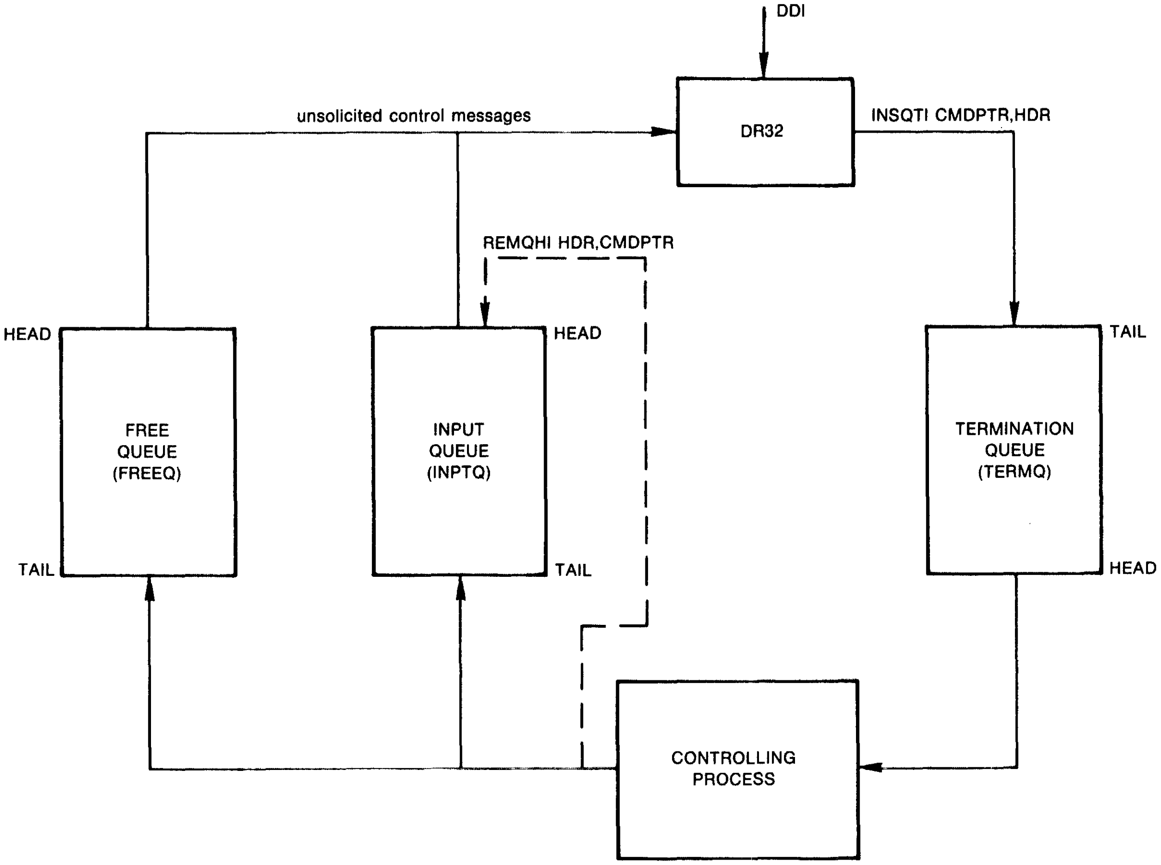
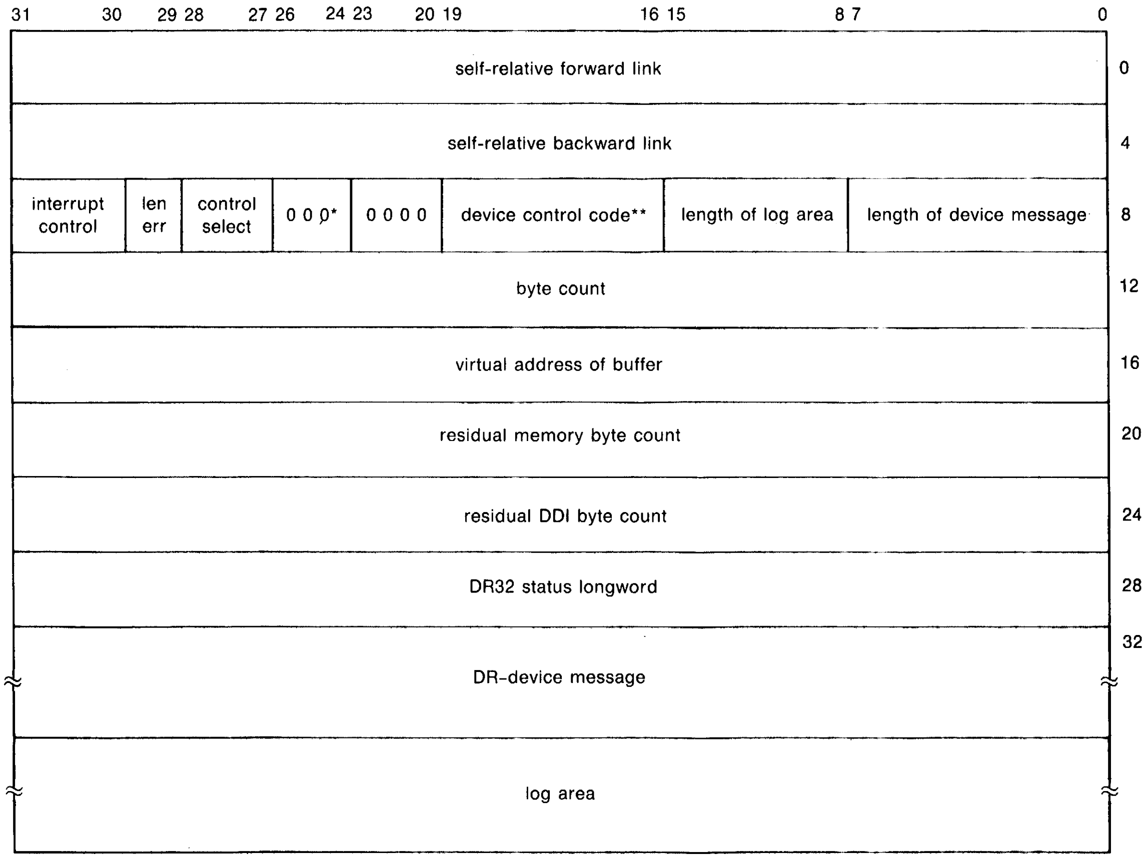


Figure 11-4 DR32 Command Packet Queue Flow

11.4.3 Command Packets

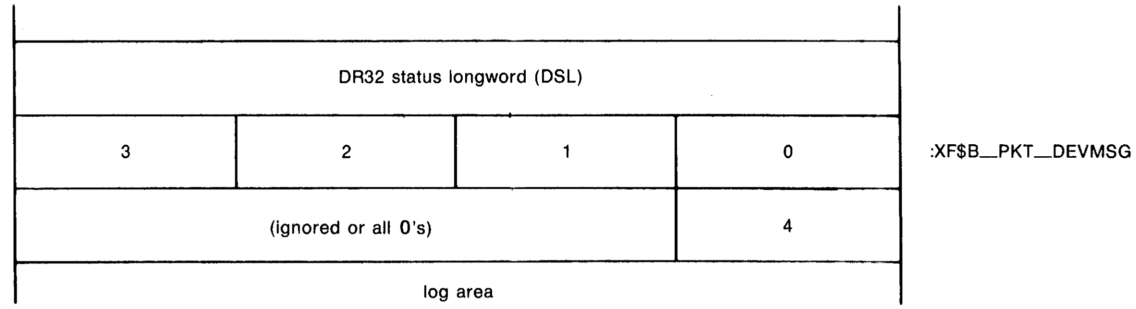
To provide for direct communication between the controlling process and the DR32, the DR32 fetches commands from user-constructed command packets located in main memory. Command packets contain commands for the DR32, such as the direction of transfer, and/or messages to be sent to the far end DR-device. The DR32 is simply the conveyer of these messages; it does not examine or add to their content. The controlling process builds command packets and manipulates the three queues, using the four VAX-11 self-relative queue instructions. Figure 11-5 shows the contents of a DR32 command packet.



* Bits 31:24 = Packet Control Byte
**Bits 23:16 = Command Control Byte

Figure 11-5 DR32 Command Packet

11.4.3.1 **Length of Device Message Field** - This field describes the length of the DR-device message in bytes. The message length must be less than 256 bytes. Note, however, that the length of device message field itself must always be an integral number of quadwords long. For example, if the application program requires a 5-byte device message, it must write a 5 in the length of device message field, but allocate 8 bytes for the device message field itself. In this case, the last three bytes of the field are ignored by the DR32 when transmitting a message, or written as zeros when receiving a message:



The symbolic offset for the length of device message field is XF$B_PKT_MSGLEN.

11.4.3.2 **Length of Log Area Field** - This field describes the length of the log area in bytes. The length specified must be less than 256 bytes. Note, however, that the length of log area field itself must be an integral number of quadwords long. For example, if the application program requires a 5-byte log area field, it must write a 5 in the length of log area field, but allocate 8 bytes for the log area field itself. In this case, the last three bytes of the field are written as zeros when receiving a log message (log messages are always received). The symbolic offset for the length of log area field is XF$B_PKT_LOGLEN.

11.4.3.3 **Device Control Code Field** - The device control field describes the function performed by the DR32. The field occupies the lower half of the command control byte (bits 16 through 23). VAX/VMS defines the following values:

Symbol	**Value**	**Function**
XF$K_PKT_RD	0	Read device
XF$K_PKT_RDCHN	1	Read device chained
XF$K_PKT_WRT	2	Write device
XF$K_PKT_WRTCHN	3	Write device chained
XF$K_PKT_WRTCM	4	Write device control message
	5	(reserved)
XF$K_PKT_SETTST	6	Set self-test
XF$K_PKT_CLRTST	7	Clear self-test
XF$K_PKT_NOP	8	No-op
XF$K_PKT_DIAGRI	9	Diagnostic read internal
XF$K_PKT_DIAGWI	10	Diagnostic write internal
XF$K_PKT_DIAGRD	11	Diagnostic read DDI
XF$K_PKT_DIAGWC	12	Diagnostic write control message
XF$K_PKT_SETRND	13	Set random enable
XF$K_PKT_CLRRND	14	Clear random enable
XF$K_PKT_HALT	15	Set HALT

Table 11-2 describes the functions performed by the different device control codes.

Table 11-2
Device Control Code Descriptions

Function	Meaning
Read Device	This function specifies a data transfer from the far end DR-device to the DR32. The control select field (see Section 11.4.3.4) describes the information to be transferred prior to the initiation of the data transfer.
Read Device Chained	This function specifies a data transfer from the far end DR-device to the DR32. The DR32 data chains to the buffer specified in the next command packet in INPTQ. A command packet that specifies read device chained must be followed by a command packet that specifies either read device chained or read device. All other device control codes cause an abort. If read device chained is specified, the chain continues. However, if read device is specified, that command packet is the last packet in the chain.
Write Device and Write Device Chained	These functions specify data transfers from the DR32 to the far end DR-device. Otherwise, they are similar to Read Device and Read Device Chained.
Write Device Control Message	This function specifies the transfer of a control message to the far end DR-device. This message is contained in the device message field of this command packet. The Write Device Control Message function directs the controlling DR32 to ignore the byte count and virtual address fields in this command packet.
Set Self-Test	This function directs the DR32 to set an internal self test flag and to set a disable signal on the DDI. This signal informs the far end DR-device that the DR32 is in self-test mode. In this condition the DR32 can no longer communicate with the far end DR-device.
Clear Self-Test	This function directs the DR32 to clear the internal self test flag set by the Set Self Test function and return to the normal mode of operation.
No Operation	The NOP function specifically does nothing.

(continued on next page)

Table 11-2 (Cont.)
Device Control Code Descriptions

Function	Meaning
Diagnostic Read Internal	This function directs the DR32 to fill the memory buffer, which is described by the virtual address and byte count specified in the current command packet, with the data that is stored in the DR32 data silo. The buffer is filled in a cyclic manner. For example, on the DR780 every 128-byte section of the buffer receives the silo data. The amount of data stored in the buffer equals the DDI byte count minus the SBI byte count. The DDI byte count is equal to the original byte count. No data transmission takes place on the DDI for this function. On the DR780, the Diagnostic Read Internal function destroys the first four bytes in the silo before storing the data in the buffer.
Diagnostic Write Internal	This function, together with the Diagnostic Read Internal function, is used to test the DR32 read and write capability. The Diagnostic Write Internal function directs the DR32 to store data, which is contained in the memory buffer described by the current command packet, in the DR32 data silo, a fifo-type buffer. No data transmission takes place on the DDI for this function. The Diagnostic Write Internal function terminates when either of the following conditions occur: ● The memory buffer is empty (the SBI byte count is 0). ● An abort has occurred. At the time the function terminates, the amount of data in the silo equals the DDI byte count minus the SBI memory byte count (Sections 11.4.3.9 and 11.4.3.10 describe these values).
Diagnostic Read DDI	This function tests transmissions over the data portion of the DDI. The DR32 must be in the self-test mode. If not, an abort will occur. On the DR780, the Diagnostic Read DDI function transmits the contents of DR32 data silo locations 0 - 127 over the DDI and returns the data to the same locations. If data transmission is normal, that is, without errors, the residual memory count is equal to the original byte count, the residual DDI count is 0, and the contents of the silo remain unchanged.

(continued on next page)

Table 11-2 (Cont.)
Device Control Code Descriptions

Function	Meaning
Diagnostic Write Control Message	This function tests transmissions over the control portion of the DDI. The DR32 must be in self-test mode. If not, an abort will occur. The Diagnostic Write Control Message function directs the DR32 to remove the command packet on FREEQ and check the length of message field. Then the first byte of the message in the command packet on INPTQ is transmitted and read back on the control portion of the DDI. This byte is then written into the message space of the packet from FREEQ. The updated packet from FREEQ is inserted onto TERMQ and is followed by the packet from INPTQ.
Set Random Enable and Clear Random Enable	The Set Random Enable function directs the DR32 to accept read and write commands sent by the far end DR-device. Range checking is performed to verify that all addresses specified by the far end DR-device for access are within the buffer block. Far end DR-device initiated transfers to or from the VAX-11 memory are conducted without notification of the VAX-11 processor or the application program. The Clear Random Enable function directs the DR32 to reject far end DR-device initiated transfers. Random access mode must be enabled when the DR32 is used in a processor-to-processor link.
Set HALT	This function places the DR32 in a halt state. The Set Halt function always generates a packet interrupt regardless of the value in the interrupt control field (see Section 11.4.3.6). If an AST routine was requested on completion of the QIO function (see Sections 11.4.5.2 and 11.4.6.2), the routine is called after the command packet containing the Set HALT function has been processed by the DR32.

The following symbolic offsets are defined for the device control code field:

Symbol	Meaning
XF$B_PKT_CMDCTL	Byte offset from the beginning of the command packet
XF$V_PKT_FUNC	Bit offset from XF$B_PKT_CMDCTL
XF$S_PKT_FUNC	Size of this bit field

11.4.3.4 **Control Select Field** - This field describes what part of the command packet will be transmitted to the far end DR-device. The control select field is examined only for the read device, read device chained, write device, and write device chained functions; for all others, it is ignored. VAX/VMS defines the following values:

Symbol	Value	Function
XF$K_PKT_NOTRAN	0	No transmission. Nothing is transmitted over the control portion of the DDI. However, if the command packet specifies a data transfer, data can be transmitted over the data portion of the DDI. The primary use of this code is during data chaining.
XF$K_PKT_CB	1	Command control byte (bits 23:16) only. This code directs the DR32 to transmit the contents of the command control byte, which includes the device control code field, to the far end DR-device. This code is used primarily at the start of data chains or nondata chain commands.
XF$K_PKT_CBDM	2	Command control byte and device message. This code directs the DR32 to transmit the command control byte, and then the device message. The primary use of this code is when an interface requires more than one byte of command.
XF$K_PKT_CBDMBC	3	Command control byte, device message, and byte count. This code directs the DR32 to transmit the command control byte, the device message, and the byte count (in that order). The primary use of this code is during processor-to-processor link operations. In this case the device message must be exactly four bytes in length and contain the virtual address of the buffer in the far end processor's memory.

The following symbolic offsets are defined for the control select field:

Symbol	Meaning
XF$B_PKT_PKTCTL	Byte offset from the beginning of the command packet
XF$V_PKT_CISEL	Bit offset from XF$B_PKT_PKTCTL
XF$S_PKT_CISEL	Size of this bit field

11.4.3.5 **Suppress Length Error Field** - This function prevents the DR32 from aborting if the data transfer on the DDI is terminated by the far end DR-device before the DDI byte counter has reached zero.

The following symbolic offsets are defined for the suppress length error field:

Symbol	Meaning
XF$B_PKT_PKTCTL	Byte offset from the beginning of the command packet
XF$V_PKT_SLNERR	Bit offset from XF$B_PKT_PKTCTL
XF$S_PKT_SLNERR	Size of this bit field

11.4.3.6 **Interrupt Control Field** - This field determines the conditions under which an interrupt is generated, on a packet by packet basis, when the DR32 places this command packet onto TERMQ. Depending on the conditions specified in the IO$_STARTDATA call, the interrupt can set an event flag and/or call an AST routine.

Symbol	Value	Function
XF$K_PKT_UNCOND	0	Interrupt unconditionally
XF$K_PKT_TMQMT	1	Interrupt only if TERMQ was previously empty
XF$K_PKT_NOINT	2,3	No interrupt

If the function is Set Halt, this field is ignored. The Set Halt function unconditionally causes a packet interrupt. The following symbolic offsets are defined for the interrupt control field:

Symbol	Meaning
XF$B_PKT_PKTCTL	Byte offset from the beginning of the command packet
XF$V_PKT_INTCTL	Bit offset from XF$B_PKT_PKTCTL
XF$S_PKT_INTCTL	Size of this bit field

11.4.3.7 **Byte Count Field** - This field specifies the size in bytes of the data buffer for this data transfer. Together with the virtual address of buffer field, this field describes the buffer in the buffer block that the DR32 will read from or write into.

The following symbolic offset is defined for the byte count field:

Symbol	Meaning
XF$B_PKT_BFRSIZ	Byte offset from the beginning of the command packet

11.4.3.8 **Virtual Address of Buffer Field** - This field specifies the virtual address of the data buffer for this data transfer. Together with the byte count field, this field describes the buffer in the buffer block that the DR32 will read from or write into.

The following symbolic offset is defined for the virtual address of buffer field:

Symbol	Meaning
XF$B_PKT_BFRADR	Byte offset from the beginning of the command packet

11.4.3.9 Residual Memory Byte Count Field

- After completion of a read device, read device chained, write device, write device chained, diagnostic read internal, diagnostic write internal, or diagnostic read DDI command specified in this command packet, the DR32 places the packet onto TERMQ for return to the controlling process. At that time, this field will contain a byte count. The difference between the count specified in the byte count field and the count in this field represents the number of bytes transferred to or from main memory, depending on the direction of transfer.

The following symbolic offset is defined for the residual memory byte count field:

Symbol	Meaning
XF$L_PKT_RMBCNT	Byte offset from the beginning of the command packet

(See also the descriptions of the Diagnostic Read Internal and Diagnostic Write Internal functions in Table 11-2.)

11.4.3.10 Residual DDI Byte Count Field

- After completion of a read device, read device chained, write device, write device chained, diagnostic read internal, diagnostic write internal, or diagnostic read DDI command specified in this command packet, the DR32 places the packet onto TERMQ for return to the controlling process. At that time, this field contains a byte count. The difference between the count specified in the byte count field and the count in this field represents the number of bytes transferred to or from the far end DR-device over the DDI, depending on the direction of transfer.

The following symbolic offset is defined for the residual DDI byte count field:

Symbol	Meaning
XF$L_PKT_RDBCNT	Byte offset from the beginning of the command packet

(See also the descriptions of the Diagnostic Read Internal and Diagnostic Write Internal functions in Table 11-2.)

11.4.3.11 **DR32 Status Longword** (DSL) - The DR32 stores the final status for a command packet in the DR32 status longword before inserting the packet onto TERMQ. The longword contains two distinct status fields:

31 24	23 16	15 0
0	DDI status	16 bits of status

Table 11-3 lists the names for the status bits returned in the DR32 status longword.

Table 11-3
DR32 Status Longword (DSL) Status Bits

Name	Meaning
	16 bits of status
XF$V_PKT_SUCCESS XF$M_PKT_SUCCESS	If set, the command was performed successfully. If not set, one of the following bits must be set: XF$M_PKT_INVPTE XF$M_PKT_RNGERR XF$M_PKT_UNGERR XF$M_PKT_INVPKT XF$M_PKT_FREQMT XF$M_PKT_DDIDIS XF$M_PKT_INVDDI XF$M_PKT_LENERR XF$M_PKT_DRVABT XF$M_PKT_PARERR XF$M_PKT_DDIERR
XF$V_PKT_CMDSTD XF$M_PKT_CMDSTD	If set, the command specified in this packet was started.
XF$V_PKT_INVPTE XF$M_PKT_INVPTE	If set, the DR32 accessed an invalid page table entry.
XF$V_PKT_FREQPK XF$M_PKT_FREQPK	If set, this command packet was removed from FREEQ.
XF$V_PKT_DDIDIS XF$M_PKT_DDIDIS.	If set, the far end DR-device is disabled.
XF$V_PKT_SLFTST XF$M_PKT_SLFTST.	If set, the DR32 is in self-test mode.
XF$V_PKT_RNGERR XF$M_PKT_RNGERR	Range error. If set, a user-provided. address was outside the command block or buffer block.

(continued on next page)

Table 11-3 (Cont.)
DR32 Status Longword (DSL) Status Bits

Name	Meaning
XF$V_PKT_UNQERR XF$M_PKT_UNQERR	If set, a queue element was not aligned on a quadword boundary.
XF$V_PKT_INVPKT XF$M_PKT_INVPKT	If set, this packet was not a valid DR32 command packet.
XF$V_PKT_FREQMT XF$M_PKT_FREQMT	If set, a message was received from the far end DR-device and FREEQ was empty.
XF$V_PKT_RNDENB XF$M_PKT_RNDENB	If set, random access mode is enabled.
XF$V_PKT_INVDDI XF$M_PKT_INVDDI	If set, a protocol error occurred on the DDI.
XF$V_PKT_LENERR XF$M_PKT_LENERR	If set, the far end DR-device terminated the data transfer before the required number of bytes were sent, or a message was received from the far end DR-device and the device message field in the command packet at the head of FREEQ was not large enough to hold it.
XF$V_PKT_DRVABT XF$M_PKT_DRVABT	The I/O driver aborted the transfer. Usually the result of a Cancel I/O ($CANCEL) system service request.
XF$V_PKT_PARERR XF$M_PKT_PARERR	A parity error occurred on the data or control portion of the DDI.
	DDI Status
XF$V_PKT_DDISTS XF$S_PKT_DDISTS	DDI status. This field is the 1-byte DDI register 0 of the far end DR-device. The following three bits are offsets to this field.
XF$V_PKT_NEXREG XF$M_PKT_NEXREG	An attempt was made to access a non-existent register in the far end DR-device.
XF$V_PKT_LOG XF$M_PKT_LOG	The far end DR-device registers are stored in the log area.
XF$V_PKT_DDIERR XF$M_PKT_DDIERR	An error occurred on the far end DR-device.

11.4.3.12 **Device Message Field** - This field contains control information to be sent to the far end DR-device. It is used when more than one byte of command is required. The number of bytes in the device message is specified in the length of device message field (see Section 11.4.3.1). (The number of bytes allocated for the length of device message field must be rounded up to an integral number of quadwords.)

If the far end DR-device is a DR32 that is connected to another processor, a device message can be sent only if the function specified in the device control code field of this command packet is read device, read device chained, write device, write device chained, or write device control message.

In the case of a write device control message, the data in the device message field is treated as unsolicited input and written into the device message field of a command packet taken from the far end DR32's FREEQ.

In the case of a read or write (either chained or unchained) function, the only message allowed is the address of the buffer in the far end processor that either contains or will receive the data to be transferred. This device message must be exactly four bytes in length. In this case the device message is not stored in the command packet from the far end DR32's FREEQ, but is used by the far end DR32 to perform the data transfer.

The device message field is also used in command packets placed on FREEQ to convey unsolicited control messages from the far end DR-device.

The symbolic offset for the device message field is XF$B_PKT_DEVMSG.

11.4.3.13 **Log Area Field** - This field receives the return status and other information from the far end DR-device's DDI registers. Logging must be initiated by the far end DR-device. The presence of a log area does not automatically cause logging to occur.

If the DR32 is connected in a processor-to-processor configuration, the log area field is not used.

11.4.4 DR32 Microcode Loader

The DR32 microcode loader program XFLOADER must be executed prior to using the DR32. Running XFLOADER requires CMKRNL and LOG_IO privileges. Typically, a command to run XFLOADER is placed in the site-specific system starting file. XFLOADER locates the file containing the DR32 microcode in the following manner:

1. XFLOADER attempts to open a file using the logical name XFc$WCS, where "c" is the DR32 controller designator. For example, to load microcode on device XFA0, XFLOADER attempts to open a file with the logical name XFA$WCS.

2. If the opening procedure described in Step 1 fails, XFLOADER attempts to open the file SYS$SYSTEM:XF780.ULD which is the default location and filename for the DR780 microcode.

After loading microcode into all available DR32s, XFLOADER either exits or hibernates, according to the following:

- If XFLOADER was run with an ordinary RUN command, that is, RUN XFLOADER, it exits after loading microcode.
- If XFLOADER was run as a separate process, as with the command

  ```
  RUN/UIC=[1,1]/PROCESS=XFLOADER SYS$SYSTEM:XFLOADER
  ```

  then it hibernates after loading microcode. In this case, XFLOADER automatically reloads microcode into the DR32s after a power recovery.

XFLOADER performs a load microcode QIO to the DR32 driver.

11.4.5 DR32 I/O Function Codes

The DR32 I/O functions are:

- Load microcode into the DR32.
- Start a DR32 data transfer.

Normally, the controlling process stops data transfers with a Set HALT command packet. However, the Cancel I/O on Channel ($CANCEL) system service can be used to abort data transfers and complete the I/O operation.

11.4.5.1 **Load Microcode** - This I/O function resets the DR32 and loads an image of DR32 microcode. The load microcode function also sets the DR32 data rate to the last specified value. Physical I/O privilege is required. VAX/VMS defines a single function code:

IO$_LOADMCODE - load microcode

The load microcode function takes two device/function-dependent arguments:

- P1 = the starting virtual address of the microcode image that is to be loaded into the DR32
- P2 = the number of bytes to be loaded (maximum of 5120 for the DR780)

If any data transfer requests are active at the time a load microcode request is issued, the load request is rejected and SS$_DEVACTIVE is returned in the I/O status block.

The microcode is verified by addressing each microword and checking for a parity error. (The microcode is not compared to the buffer image.) If there are no parity errors, then the microcode was loaded successfully and the driver sets the microcode valid bit in one of the DR32 registers. If there is a parity error, SS$_PARITY is returned in the I/O status block. (The valid bit is cleared by the reset operation.)

In addition to SS$_PARITY, three other status codes can be returned in the I/O status block: SS$_NORMAL, SS$_DEVACTIVE, and SS$_POWERFAIL.

11.4.5.2 **Start Data Transfer** - This function specifies a command table that holds the parameters required to start the DR32. In addition to several other parameters, the command table contains the size and address of the command and buffer blocks, and the address of a command packet AST routine. No user privilege is required. VAX/VMS defines a single function code:

IO$_STARTDATA - start data transfer

The start data transfer function takes one function modifier:

IO$M_SETEVF - set event flag

If IO$M_SETEVF is included with the function code, the specified event flag is set whenever a command packet interrupt occurs, and when the start data transfer QIO is completed. If IO$M_SETEVF is not specified, the event flag is set only when the QIO is completed.

IO$M_SETEVF should not be used with the $QIOW macro because the $QIOW will return after the event flag is set the first time.

The start data transfer function takes two device/function-dependent arguments:

- P1 = the starting virtual address of the Data Transfer Command Table in the user's process
- P2 = the length in bytes (always 32) of the Data Transfer Command Table. (The symbolic name is XF$K_CMT_LENGTH.)

The format of the Data Transfer Command Table is shown in Figure 11-6 (offsets are shown in parentheses).

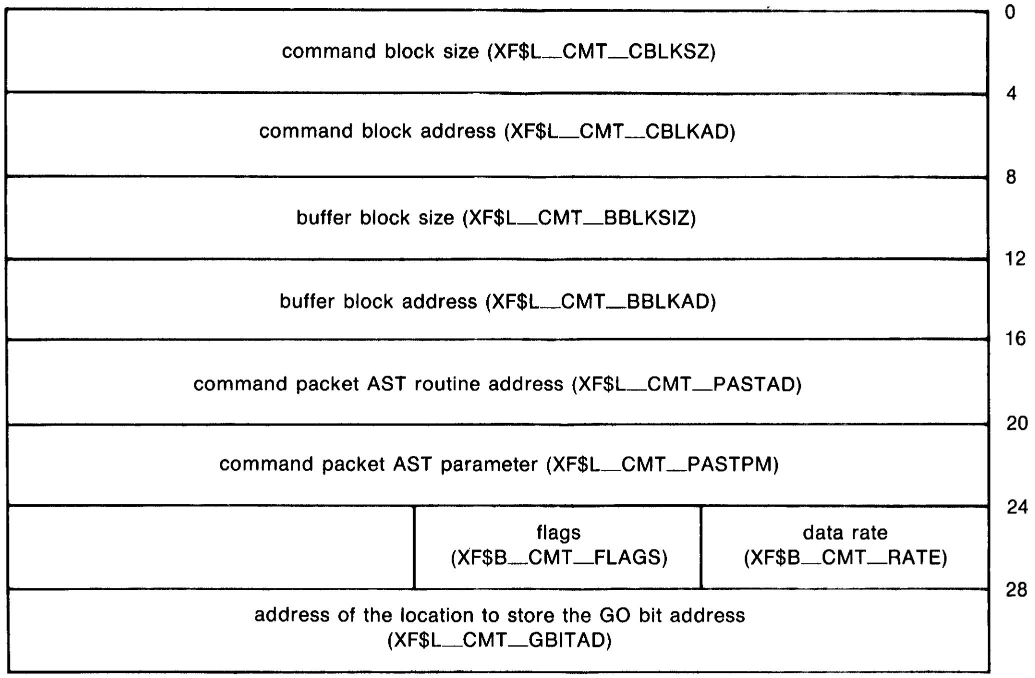

0
command block size (XF$L_CMT_CBLKSZ)
4
command block address (XF$L_CMT_CBLKAD)
8
buffer block size (XF$L_CMT_BBLKSIZ)
12
buffer block address (XF$L_CMT_BBLKAD)
16
command packet AST routine address (XF$L_CMT_PASTAD)
20
command packet AST parameter (XF$L_CMT_PASTPM)
24
flags (XF$B_CMT_FLAGS)
data rate (XF$B_CMT_RATE)
28
address of the location to store the GO bit address (XF$L_CMT_GBITAD)

Figure 11-6 Data Transfer Command Table

Since the command block contains the queue headers for INPTQ, TERMQ, and FREEQ, its address in the second longword must be quadword aligned.

The command packet AST routine specified in the fifth longword is called whenever the DR32 signals a command packet interrupt. A command packet AST should be distinguished from a QIO AST (astadrs argument). A command packet interrupt occurs whenever the DR32 completes a function and returns a packet that specifies an interrupt (see Section 11.4.3.6) by inserting it onto TERMQ. The astadrs argument address is called when the QIO is completed. If either the command packet AST address or the astadrs address is 0, the respective AST is not delivered. If the command packet specifies the Set HALT function, a command packet interrupt occurs regardless of the state of the packet interrupt bits.

The seventh longword contains the data rate byte and a flags byte. The data rate byte controls the DR32 clock rate. The data rate value is considered to be an unsigned integer.

For the DR780, the relationship between the value of the data rate byte and the actual data rate is given by the following formula:

$$\text{Data rate (in megabytes/sec)} = \frac{40}{(256 - \text{value of data rate byte})}$$

For example, a data rate value of 236 corresponds to an actual data rate of 2.0 Megabytes/sec. Note that the DR780 ignores data rate values greater than 251.

The parameter XFMAXRATE set at system generation limits the maximum data rate that can be set. This parameter limits the maximum data rate because very high data rates on certain configurations can cause a processor timeout. If the user attempts to set the data rate higher than the rate allowed by XFMAXRATE, the error status SS$_BADPARAM is returned in the I/O status block.

VAX/VMS defines the following flag bit values:

XF$V_CMT_SETRTE	If set, XF$B_CMT_RATE specifies the data rate. If clear, the data rate established by a previous $IO_STARTDATA QIO is used. The IO$_LOADMCODE function sets the data rate to the last value used. If the data rate has not been previously set, a value of 0 is used.
XF$V_CMT_DIPEAB	If set, parity errors on the data portion of the DDI do not cause device aborts. If clear, a parity error results in a device abort.

The eighth longword contains the address of a location to store the address of the GO bit. This bit must be set whenever the application program inserts a command packet onto an empty INPTQ. The GO bit register is mapped in system memory space and the address is returned to the user.

The IO$_STARTDATA function locks the command and buffer blocks into memory and starts the DR32. Whenever the DR32 interrupts with a command packet interrupt, the driver queues a packet AST (if an AST address is specified) and, if IO$M_SETEVF is specified, sets the event flag. The QIO remains active until one of the following events occur:

1. A Set HALT command packet is processed by the DR32.
2. The data transfer aborts.
3. A Cancel I/O ($CANCEL) system service is issued on this channel.

If an abort occurs, the second longword of the I/O status block contains additional bits that identify the cause of the abort (see Section 11.5).

The start data transfer function can return twelve error codes in the I/O status block: SS$_BUFNOTALIGN, SS$_CTRLERR, SS$_ABORT, SS$_CANCEL, SS$_EXQUOTA, SS$_INSFMEM, SS$_MCNOTVALID, SS$_NORMAL, SS$_IVBUFLEN, SS$_DEVREQERR, SS$_PARITY, and SS$_POWERFAIL.

11.4.6 **High-level Language Interface**

VAX/VMS supports a set of program-callable procedures that provide access to the DR32. The formats of these calls are documented here for VAX-11 FORTRAN users. VAX-11 MACRO users must set up a standard VAX/VMS argument block and issue the standard procedure CALL. (Optionally, VAX-11 MACRO users can access the DR32 directly by issuing a IO$_STARTDATA QIO, building command packets, and inserting them onto INPTQ.) Users of other high-level languages can also specify the proper subroutine or procedure invocation.

VAX/VMS provides six high-level language procedures for the DR32. They are contained in the default system library, STARLET.OLB. Table 11-4 lists these procedures. Procedure arguments are either input or output arguments, that is, arguments supplied by the user or arguments that will contain information stored by the procedure. Except for those that are indicated as output arguments, all arguments in the following call descriptions are input arguments. By default, all procedure arguments are integer variables unless otherwise indicated.

VAX/VMS high-level language support routines for the DR32 do the following:

- Issue QIOs
- Allocate and manage the command memory
- Build command packets, insert them onto INPTQ, and set the GO bit
- Remove command packets from TERMQ and return the information they contain to the controlling process
- Use ACTION routines for program - device synchronization

Table 11-4
VAX-11 Procedures for the DR32

Subroutine	Function
XF$SETUP	Defines command and buffer areas; initializes queues
XF$STARTDEV	Issues a QIO that starts the DR32
XF$FREESET	Releases command packets onto FREEQ
XF$PKTBLD	Builds command packets; releases them onto INPTQ
XF$GETPKT	Removes a command packet from TERMQ
XF$CLEANUP	Deassigns the device channel and deallocates the command area

VAX/VMS also provides a FORTRAN parameter file, SYS$LIBRARY:XFDEF.FOR, that can be included in FORTRAN programs. This file defines many (but not all) of the XF$... symbolic names described in this chapter. For example, SYS$LIBRARY:XFDEF.FOR contains symbolic definitions for function codes (that is, device control codes), interrupt control codes, command control codes, and masks for error bits set in the I/O status block and the DR32 Status Longword. To include these definitions in a FORTRAN program, insert the following statement in the source code:

```
INCLUDE 'SYS$LIBRARY:XFDEF.FOR'
```

11.4.6.1 **XF$SETUP** - The XF$SETUP subroutine defines memory space for the command and buffer areas, and initializes INPTQ, TERMQ, and FREEQ. The call to XF$SETUP must be made prior to any calls to other DR32 support routines.

The format of the XF$SETUP call is as follows:

```
CALL XF$SETUP(contxt,barray,bufsiz,numbuf,[idevmsg],[idevsiz],
         [ilogmsg],[ilogsiz],[cmdsiz],[status])
```

Argument descriptions are as follows:

contxt A 30-longword user-supplied array that is maintained by the support routines and is used to contain context and status information concerning the current data transfer (see Section 11.4.6.5). The contxt array provides a common storage area that all support routines share. For increased performance, contxt should be longword-aligned.

barray Specifies the starting virtual address of an array of buffers that, in the case of an output operation contain information for transfer by the DR32, or in the case of an input operation, will contain information transferred by the DR32. For example, if barray is declared INTEGER*2 BARRAY (I,J), I is the size of each data buffer in words and J is the number of buffers. The lower bound on both indices is assumed to be 1. All buffers in the array must be contiguous to each other and of fixed size.

bufsiz Specifies the size in bytes of each buffer in the array. All buffers are the same size. If the barray argument is declared as stated above, bufsiz = I*2. The bufsiz argument length is one longword.

numbuf Specifies the number of buffers in the array. If the barray argument is declared as in the preceding paragraph, numbuf = J. The area of memory described by the barray, bufsiz, and numbuf arguments is used as the buffer block for DR32 data transfers. The numbuf argument length is one longword.

idevmsg Specifies an array, declared by the application program, that is used to store an unsolicited input device message from the far end DR-device. The DR32 stores unsolicited input in the device message field of a command packet from FREEQ and places that packet onto TERMQ. When XF$GETPKT removes such a packet from TERMQ, it copies the device message field into the idevmsg array. The calling program is then notified that information has been stored in the idevmsg array. The idevmsg argument is optional; the argument must be given if any unsolicited input is anticipated.

idevsiz Specifies the size in bytes of the idevmsg array. The maximum size of a device message is 256 bytes. The idevsiz argument is optional; if idevmsg is specified, idevsiz must be specified. The idevsiz argument length is one word.

ilogmsg Specifies an array, declared by the application program, that is used to store log information from the far end DR-device contained in the log area field of the command packet. Log information is hardware-dependent data that is returned by the far end DR-device. The XF$SETUP routine stores the address and size of the ilogmsg array; the log information is stored in the ilogmsg array by the XF$GETPKT routine. The ilogmsg argument is optional; the argument must be given if any log information is anticipated.

ilogsiz Specifies the size in bytes of the ilogmsg array. The maximum size of a log message is 256 bytes. The ilogsiz argument is optional. However, if ilogmsg is specified, ilogsiz must be specified. The ilogsiz argument length is one word.

cmdsiz Specifies the amount of memory space to be allocated from which command packets are to be built. The user must consider the following factors when deciding how much memory to allocate for this purpose:

1. The number of command packets that the application program will be using.

2. That the device message and log area fields in command packets are rounded up to quadword boundaries.

3. That the size of the command packet itself is rounded up to an 8-byte boundary.

4. That cmdsiz will be rounded up to a page boundary.

The cmdsiz argument is optional; argument length is one longword. If defaulted, the allocated space is equal to:

(numbuf)*(32+idevsiz+ilogsiz)*(3)

which is rounded up to a full page.

Memory space for command packets is obtained by calling LIB$GET_VM.

status This output argument receives the VAX/VMS success or failure code of the XF$SETUP call:

SS$_NORMAL	Normal successful completion
SS$_BADPARAM	Invalid input argument
Error returns from LIB$GET_VM	

The status argument is optional; argument length is one longword.

11.4.6.2 **XF$STARTDEV** - The XF$STARTDEV subroutine issues the QIO request that starts the DR32 data transfer.

The format of the XF$STARTDEV call is as follows:

```
CALL XF$STARTDEV(contxt,devnam,[pktast],[astparm],[efn],[modes],
        [datart],[status])
```

Argument descriptions are as follows:

contxt Specifies the array that contains context and status information (see Section 11.4.6.1).

devnam Specifies the device name (logical name or actual device name) of the DR32. All letters in the resultant string must be capitalized and the device name must terminate with a colon, for example, "XFA0:". The devnam datatype is character string.

pktast Specifies the address of an AST routine that is called each time a command packet that specifies an interrupt in its interrupt control field is returned by the DR32, that is, placed onto TERMQ (see Section 11.4.7.2). This AST routine is also called on completion of the QIO request. Normally, the AST routine would call XF$GETPKT to remove command packets from TERMQ until TERMQ is empty. The pktast argument is optional.

astparm Specifies a longword parameter that is included in the call to the pktast-specified AST routine. The format used to call the AST routine is:

```
CALL pktast(astparm)
```

The astparm argument is optional; argument length is one longword. If astparm is not specified, pktast is called with no parameter.

efn If the event flag must be determined by the application program, efn specifies the number of the event flag that is set when a packet interrupt is delivered. Otherwise, it is not necessary to include this argument in a XF$STARTDEV call. If defaulted, efn is 21. The efn argument length is one word.

The event flag (either the default or the event flag specified by this argument) is set for every packet interrupt, and also when the QIO completes.

modes Specifies the mode of operation. VAX/VMS defines the following value:

2 = parity errors on the data portion of the DDI do not cause the device to abort.

If defaulted, modes is 0 (a parity error causes the device to abort)

datart Specifies the data rate. The data rate controls the speed at which the transfer takes place. The data rate is considered to be an unsigned integer in the range 0 to 255. The relationship between the specified data rate value and the actual data rate is given by the following formula:

$$\text{Data rate} = \atop \text{(in megabytes/sec)} \quad \frac{40}{\text{(256 - value of data rate byte)}}$$

For example, a data rate value of 236 corresponds to an actual data rate of 2.0 megabytes/sec. Note that the DR780 ignores data rate values greater than 251.

If datart is defaulted, the previously set data rate is used. The datart argument length is one byte.

status This output argument receives the VAX/VMS success or failure code of the XF$STARTDEV call:

SS$_NORMAL	Normal successful completion
SS$_BADPARAM	Required parameter defaulted

Error returns from $CREATE (which is called to assign a channel to the device) and $QIO

The status argument is optional; argument length is one longword.

11.4.6.3 **XF$FREESET** - The XF$FREESET subroutine releases command packets onto FREEQ. These packets are then available to the DR780 to store any unsolicited input from the far end DR-device. If unsolicited input from the far end DR-device is expected, the XF$FREESET call should be made before the XF$STARTDEV call is issued.

Idevsiz, the argument that specifies the size of the idevmsg array in the call to XF$SETUP, defines the size of the device message field in command packets inserted onto FREEQ. This is because unsolicited device messages are copied from the device message field of the command packet to the idevmsg array.

Note that the XF$FREESET subroutine may occasionally disable ASTs for a very short period.

The format of the XF$FREESET call is as follows:

```
    CALL XF$FREESET(contxt,[numpkt],[intctrl],[action],[actparm],
                 [status])
```

Argument descriptions are as follows:

contxt — Specifies the array that contains context and status information (see Section 11.4.6.1).

numpkt — Specifies the number of command packets to be released onto FREEQ. The numpkt argument is optional; argument length is one word. If defaulted, numpkt is 1.

intctrl — Specifies the conditions under which an AST is delivered (and the event flag set) when the DR32 places this command packet (or packets) on TERMQ (see Section 11.4.6.2). VAX/VMS defines the following values:

0 = unconditional AST delivery and event flag set
1 = AST delivery and event flag set only if TERMQ is empty
2 = no AST interrupt or event flag set

The intctrl argument is optional; argument length is one word. If defaulted, intctrl is 0.

action — Specifies the address of a routine that is called when any command packet built by this call to XF$FREESET is removed from TERMQ by XF$GETPKT (see Section 11.4.7.3). The action argument is optional.

actparm — A longword parameter that is passed to the action routine when the action routine is called (see Section 11.4.7.3). The actparm argument is optional.

status   This output argument receives the VAX/VMS success or failure code of the XF$FREESET call:

SS$_NORMAL	Normal successful completion
SS$_BADQUEUEHDR	FREEQ interlock timeout
SS$_INSFMEM	Insufficient memory to build command packets
SHR$_NOCMDMEM	Command memory is not allocated (usually because the data transfer has stopped and XF$CLEANUP has been called, or because XF$SETUP has not been called)

11.4.6.4 **XF$PKTBLD** - The XF$PKTBLD subroutine builds command packets and releases them onto INPTQ.

Note that the XF$PKTBLD subroutine may occasionally disable ASTs for a very short period.

The format of the XF$PKTBLD call is as follows:

```
CALL XF$PKTBLD(contxt,func,[index],[size],[devmsg],[devsiz],
        [logsiz],[modes],[action],[actparm],[status])
```

Argument descriptions are as follows:

contxt   Specifies the array that contains context and status information (see Section 11.4.6.1).

func   Specifies the device control code. Device control codes describe the function the DR32 is to perform. The func argument length is one word. VAX/VMS defines the following values (Table 11-2 describes the functions in greater detail):

Symbol	**Value**	**Function**
XF$K_PKT_RD	0	Read device
XF$K_PKT_RDCHN	1	Read device chained
XF$K_PKT_WRT	2	Write device
XF$K_PKT_WRTCHN	3	Write device chained
XF$K_PKT_WRTCM	4	Write device control message
	5	(reserved)
XF$K_PKT_SETTST	6	Set self-test
XF$K_PKT_CLRTST	7	Clear self-test
XF$K_PKT_NOP	8	No-op
XF$K_PKT_DIAGRI	9	Diagnostic read internal
XF$K_PKT_DIAGWI	10	Diagnostic write internal
XF$K_PKT_DIAGRD	11	Diagnostic read DDI
XF$K_PKT_DIAGWC	12	Diagnostic write control message
XF$K_PKT_SETRND	13	Set random enable
XF$K_PKT_CLRRND	14	Clear random enable
XF$K_PKT_HALT	15	Set HALT

index — Specifies the index of a data buffer specified by the barray argument (see Section 11.4.6.1). The specific index value given means that elements barray (1,index) through barray (size,index) will be transferred, that is, one buffer full of data. The index argument is optional and only used when the function specifies a data transfer, that is, a read device, read device chained, write device, or write device chained function. The index argument length is one word.

size — Specifies a byte count to be transferred. This argument is optional and only used when the function specifies a data transfer. If defaulted, the number of bytes to be transferred is assumed to be the size of the buffer (specified by the bufsiz argument in the call to XF$SETUP). If the size argument is given, then the specified number of bytes of data (barray (1,index) through barray (size,index)) will be transferred. If size is defaulted and the function specifies a data transfer, then barray (1,index) through barray (bufsiz,index) will be transferred. The size argument length is one longword.

devmsg — Specifies a variable that contains the device message to be sent to the far end DR-device. Provides additional control of the far end DR-device see Section 11.4.3.12. The devmsg argument is optional.

devsiz — Specifies the size in bytes of the devmsg variable. If the modes argument specifies that a device message is to be sent over the control portion of the DDI, devsiz specifies the number of bytes of devmsg that will be sent to the far end DR-device.

logsiz — Specifies the size of the log message expected from the far end DR-device. The logsiz argument is optional, argument length is one word. If defaulted, logsiz is 0.

modes — Provides additional control of the transaction. VAX/VMS defines the following values:

Value	Meaning
+8	Only the function code is sent over the control portion of the DDI to the far end DR-device. Only for read device, read device chained, write device, and write device chained functions.
+16	The function code and the device message are sent over the control portion of the DDI to the far end DR-device. Only for read device, read device chained, write device, and write device chained functions.

+24 The function code, the device message, and the buffer size are sent over the control portion of the DDI to the far end DR-device. Only for read device, read device chained, write device, and write device chained functions.

If none of the above three values is selected, nothing is transmitted over the control portion of the DDI to the far end DR-device.

+32 Length errors are suppressed. If not selected, a length error results in an abort.

+64 An AST should be delivered (and an event flag set) when this command packet is inserted onto TERMQ, provided TERMQ is empty.

+128 No AST is delivered or event flag set for this command packet.

If both +64 and +128 are selected, +128 takes precedence.

If neither of the above two values is selected, ASTs are delivered and the event flag is set unconditionally, that is, whenever this command packet is placed onto TERMQ.

+256 Insert this command packet at the head of INPTQ. If not selected, insert the packet at the tail of INPTQ.

The modes argument default value is 0.

action Specifies the address of a routine that is called when XF$GETPKT removes this command packet from TERMQ. This occurs after the DR32 has completed the command specified in the packet (see Section 11.4.7.3). The action argument length is one longword.

actparm A longword parameter that is passed to the action routine when the action routine is called (see Section 11.4.7.3). The actparm argument is optional.

status This output argument receives the VAX/VMS success or failure code of the XF$PKTBLD call:

SS$_NORMAL	Normal successful completion
SS$_BADPARAM	Input parameter error
SS$_BADQUEUEHDR	INPTQ interlock timeout
SS$_INSFMEM	Insufficient memory to build command packets
SHR$_NOCMDMEM	Command memory not allocated (usually because the data transfer has stopped and XF$CLEANUP has been called, or because XF$SETUP has not been called)

11.4.6.5 **XF$GETPKT** - The XF$GETPKT subroutine removes a command packet from TERMQ.

Note that the XF$GETPKT subroutine may occasionally disable ASTs for a very short period.

The format of the XF$GETPKT call is as follows:

```
CALL XF$GETPKT(contxt,[waitflg],[func],[index],[devflag],
         [logflag],[status])
```

Argument descriptions are as follows:

contxt Specifies the array that contains the context and status information (see Section 11.4.6.1). On return from XF$GETPKT, the first eight longwords of the contxt array are filled with the status of the data transfer:

I/O status block	:CONTXT
	4
control information	8
byte count	12
virtual address of buffer	16
residual memory byte count	20
residual DDI byte count	24
DR32 status longword (DSL)	28

The first two longwords are the I/O status block. The next six longwords are copied directly from bytes 8 through 31 of the command packet.

This information is returned by the DR32 as status in each command packet. With the exception of the I/O status block, the information is copied by XF$GETPKT into the contxt array whenever XF$GETPKT removes a command packet from TERMQ.

The I/O status block is stored only after the data transfer has halted and it contains the final status of the transfer. Section 11.5 describes the I/O status block.

See Section 11.4.2 for a description of the remaining fields.

waitflg Specifies the consequences of an attempt by XF$GETPKT to remove a command packet from an empty TERMQ. If waitflg is 0 (default), XF$GETPKT waits for the event flag to be set and then removes a packet from TERMQ. If waitflg is 1, XF$GETPKT returns immediately with a failure status. Normally, waitflg is set to 1 (.TRUE.) for AST synchronization and set to 0 (.FALSE.) for event flag synchronization (see Section 11.4.7). The waitflg argument is optional.

func This output argument receives the device control code specified in this command packet (see Section 11.4.6.4). The func argument is optional; argument length is one word.

index If this command packet specified a data transfer, this output argument receives the buffer index specified when this command packet was built by XF$PKTBLD (see Section 11.4.6.4). The index argument is optional; argument length is one word.

devflag If set to .TRUE. (255), this output argument indicates that a device message was stored in the idevmsg array, which is described in the XF$SETUP call (see Section 11.4.6.1). The devflag argument is optional; argument length is one byte.

logflag If set to .TRUE. (255), this output argument indicates that a log message was stored in the ilogmsg array, which is described in the XF$SETUP call (see Section 11.4.6.1). The logflag argument is optional; argument length is one byte.

status This output argument receives the status of the XF$GETPKT call:

SS$_NORMAL	Normal successful completion
SS$_BADQUEUEHDR	TERMQ interlock timeout
SHR$_QEMPTY	The TERMQ was empty but the transfer is still in progress. Only returned if waitflg is .TRUE.
SHR$_HALTED	TERMQ was empty, the transfer is complete, and the I/O status block contains the final status. XF$CLEANUP has been called automatically. Subsequent calls to XF$GETPKT return SHR$_NOCMDMEM.
SHR$_NOCMDMEM	Command memory not allocated. Usually indicates either: 1. XF$SETUP was not called. 2. XF$CLEANUP was called.

11.4.6.6 **XF$CLEANUP** - The XF$CLEANUP subroutine deassigns the channel and deallocates the command area allocated by XF$SETUP. If XF$GETPKT detects a TERMQ empty condition and the transfer has halted, it will automatically call XF$CLEANUP. However, if the transfer either terminates in a SS$_CTRLERR or SS$_BADQUEHDR error, or is intentionally terminated, XF$GETPKT may not detect these conditions and XF$CLEANUP should be called explicitly.

The format of the XF$CLEANUP call is as follows:

```
CALL XF$CLEANUP(contxt,[status])
```

Argument descriptions are as follows:

contxt Specifies the array that contains context and status information (see Section 11.4.6.1).

status This output argument receives the status of the XF$CLEANUP call:

SS$_NORMAL	Normal successful completion
SHR$_NOCMDMEM	Command memory not allocated

Error returns from LIB$FREE_VM and $DASSIGN

11.4.7 User Program - DR32 Synchronization

Synchronization of high-level language application programs with the DR32 can be achieved in three ways:

- Event flags
- AST routines
- Action routines

11.4.7.1 **Event Flags** - Event flag synchronization is attained by calling the XF$GETPKT routine (see Section 11.4.6.5) with the waitflg argument set to 0 (default). The pktast argument in the XF$STARTDEV routine (see Section 11.4.6.2) is normally defaulted. If the XF$GETPKT routine is called and the termination queue is empty, the routine waits until the DR32 places a command packet on the queue and sets the event flag. The packet is then removed from the queue and returned to the caller.

11.4.7.2 **AST Routines** - If a call to the XF$STARTDEV routine includes the pktast argument, the specified AST routine is called each time an AST is delivered. AST delivery can be controlled on a packet-by-packet basis through use of the intctrl argument in the XF$FREESET routine and by specifying appropriate values in the modes argument of the XF$PKTBLD routine (see Sections 11.4.6.3 and 11.4.6.4). For a particular command packet, ASTs can be delivered:

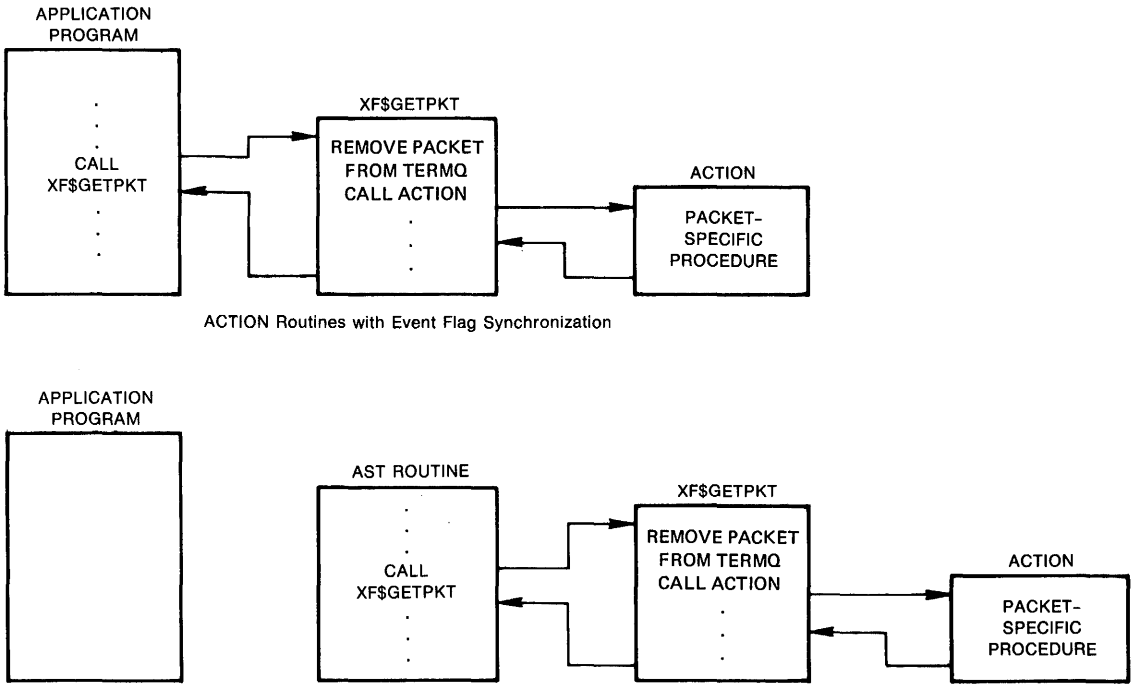
1. Unconditionally when the packet is placed onto TERMQ.
2. Only if TERMQ is empty when the packet is placed on it.
3. Not at all. That is, there is no AST when the packet is placed on TERMQ.

There is no guarantee that an AST will be delivered for every command packet, even when the astctrl argument indicates unconditional AST delivery. In particular, if packet interrupts are closely spaced, several packets may be placed onto TERMQ even though only one AST is delivered. Therefore, the AST routine should continue to call the XF$GETPKT routine until all command packets are removed from TERMQ.

11.4.7.3 **Action Routines** - The action argument specified in the XF$FREESET and XF$PKTBLD routines (see Sections 11.4.6.3 and 11.4.6.4) can be used for a more automated synchronization of the program with the DR32. Routines specified by action arguments can be used for both event flag and AST routine synchronization.

The address of the action routine is included in the command packet. This routine is automatically called by the XF$GETPKT routine when it removes that packet from TERMQ. This allows the user to define, at the time it is built, how the command packet will be handled once it is removed from TERMQ. In addition to specifying different action routines for different types of command packets, the user can also specify an action routine parameter (actparm) to further identify the command packet and/or the action to be taken on completion of the command. Figure 11-7 shows the use of action-specified routines for program synchronization.

An important difference between AST routine and action routine use is the number of times the respective routines are specified. Command packet AST routines are specified only once, in a XF$STARTDEV call; a single AST routine is implied. Action routines, however, are specified in each command packet. This allows a different action routine to be designed for each type of command packet.


APPLICATION PROGRAM
CALL XF$GETPKT
XF$GETPKT
REMOVE PACKET FROM TERMQ CALL ACTION
ACTION
PACKET-SPECIFIC PROCEDURE
APPLICATION PROGRAM
AST ROUTINE
CALL XF$GETPKT
XF$GETPKT
REMOVE PACKET FROM TERMQ CALL ACTION
ACTION
PACKET-SPECIFIC PROCEDURE

ACTION Routines with Event Flag Synchronization

ACTION Routines with AST Routine Synchronization

Figure 11-7 ACTION Routine Synchronization

Routines specified by the action argument are supplied by the user. The format of the calling interface is as follows:

```
CALL action-routine (contxt,actparm,devflag,logflag,func,
                     index,status)
```

With the exception of actparm, all arguments are the same as those described for the XF$GETPKT routine. In effect, the action routine will receive the same information XF$GETPKT optionally returns to its calling program, along with the actparm argument that was specified when the packet was built. If these variables are to be passed as inputs to the action routine, they must be supplied as output variables in the call to the XF$GETPKT routine.

11.5 I/O STATUS BLOCK

The I/O status block for the load microcode and start data transfer QIO functions is shown in Figure 11-8. The I/O status block used in the first two longwords of the contxt array for high-level language calls also employs this format.

31 27	26 24	23 16	15 0
0			status
5 status bits	0	DDI status	16 status bits

Figure 11-8 I/O Functions IOSB Content

VAX/VMS status values are returned in the first longword. Table 11-5 lists these values. If either SS$_CTRLERR, SS$_DEVREQERR, or SS$_PARITY is returned in the status word, the second longword contains additional returns, that is, device-dependent data. Table 11-6 lists these returns.

The I/O status block for a QIO function is returned after the function completes. Status is not stored on the completion of every command packet because any number of packets can pass between the application program and the DR32 during the execution of a single QIO.

Table 11-5
DR32 Status Returns

Status	Meaning
SS$_ABORT	Request aborted. A request in progress was aborted by the $CANCEL system service. (Only for start data transfer functions.)
SS$_BADPARAM	Bad parameter. An attempt was made to set the data rate higher than the rate allowed by the SYSGEN parameter XFMAXRATE. (Only for start data transfer functions.)

(continued on next page)

Table 11-5 (Cont.)
DR32 Status Returns

Status	Meaning
SS$_BADQUEHDR	Bad queue header. An INPTQ or TERMQ interlock timeout occurred.
SS$_BUFNOTALIGN	Alignment error. The command block address in the Data Transfer Command Table was not quadword aligned. (Only for start data transfer functions.)
SS$_CANCEL	Request cancelled by the $CANCEL system service before it started. (Only for the start data transfer functions.)
SS$_CTRLERR	Controller error. A fatal hardware malfunction occurred that stops all DR32 activity. (Only for start data transfer functions.) The second longword of the IOSB contains additional information pertaining to this error; the following bit values are associated with SS$_CTRLERR: XF$V_IOS_INVPTE XF$V_IOS_SBIERR XF$V_IOS_RDSERR
SS$_DEVACTIVE	Device is active. The microcode cannot be loaded because there is an active data transfer request. (Only for the load microcode function.)
SS$_DEVREQERR	DR32 user request error. A programming error or an error associated with the far end DR-device is indicated. The second longword of the I/O status block contains additional information pertaining to the error; the following bit values are associated with SS$_DEVREQERR: XF$V_IOS_DDIDIS XF$V_IOS_RNGERR XF$V_IOS_UNQERR XF$V_IOS_INVPKT XF$V_IOS_FREQMT XF$V_IOS_INVDDI XF$V_IOS_LENERR XF$V_IOS_DDIERR
SS$_EXQUOTA	AST quota exceeded. A command packet AST cannot be queued because the process AST quota was exceeded. (Only for start data transfer functions.)
SS$_INSFMEM	Insufficient dynamic memory to initiate a start data transfer request, build a command packet, or queue a command packet AST.

(continued on next page)

Table 11-5 (Cont.)
DR32 Status Returns

Status	Meaning
SS$_IVBUFLEN	Incorrect length. Either the command block size or the buffer block size is 0 or equal to or greater than 2**29, or the command table length is not XF$K_CMT_LENGTH.
SS$_MCNOTVALID	Microcode has not yet been successfully loaded or has become invalid. (Only for start data transfer functions.)
SS$_NORMAL	QIO request or support routine call completed successfully. Either the microcode was loaded successfully or the data transfer was completed successfully.
SS$_PARITY	Parity error. Either the microcode was not loaded successfully or the DR32 controller detected a parity error and a hardware malfunction is indicated. The second longword of the I/O status block contains additional information pertaining to this malfunction; the following bit values are associated with SS$_PARITY: XF$V_IOS_WCSPE XF$V_IOS_CIPE XF$V_IOS_DIPE XF$V_IOS_PARERR
SS$_POWERFAIL	A power failure occurred while a data transfer request was active or the DR32 is powered down.

Table 11-6
Device-Dependent IOSB Returns for I/O Functions

Symbolic Name	Meaning
	16 Status Bits
XF$V_PKT_SUCCESS	The command was performed successfully
XF$V_IOS_CMDSTD	Command specified in the command packet started.
XF$V_IOS_INVPTE	Invalid page table entry.
XF$V_IOS_FREQPK	This command packet came from FREEQ.
XF$V_IOS_DDIDIS	The far end DR-device is disabled.
XF$V_IOS_SLFTST	The DR32 is in self-test mode.

(continued on next page)

Table 11-6 (Cont.)
Device-Dependent IOSB Returns for I/O Functions

Symbolic Name	Meaning
XF$V_IOS_RNGERR	Range error. The user-provided address is outside the command block range or the buffer block range.
XF$V_IOS_UNQERR	A queue element was not aligned on a quadword boundary.
XF$V_IOS_INVPKT	A packet was not a valid DR32 command packet.
XF$V_IOS_FREQMT	A message was received from the far end DR-device and FREEQ was empty.
XF$V_IOS_RNDENB	Random access mode is enabled.
XF$V_IOS_INVDDI	A protocol error occurred on the DDI.
XF$V_IOS_LENERR	The far end DR-device terminated the data transfer before the required number of bytes were sent, or a message was received from the far end DR-device and the device message field in the command packet at the head of FREEQ was not large enough to hold it.
XF$V_IOS_DRVABT	The I/O driver aborted the DR32 function.
XF$V_PKT_PARERR	A parity error occurred on the data or control portion of the DDI.
	DDI Status
XF$V_IOS_DDISTS	The 1-byte status register 0 for the far end DR-device. XFV_IOS_NEXREG, XFV_IOS_LOG, and XF$V_IOS_DDIERR are returns from this register.
XF$V_IOS_NEXREG	An attempt was made to access a nonexistent register on the far end DR-device.
XF$V_IOS_LOG	The far end DR-device registers are stored in the log area.
XF$V_IOS_DDIERR	An error occurred on the far end DR-device.
	5 Status Bits
XF$V_IOS_BUSERR	An error on the processor's internal CPU memory bus occurred.
XF$V_IOS_RDSERR	A noncorrectable memory error occurred (Read Data Substitute).
XF$V_IOS_WCSPE	Writeable Control Store parity error.
XF$V_IOS_CIPE	Control Interconnect parity error. A parity error occurred on the control portion of the DDI.
XF$V_IOS_DIPE	Data Interconnect parity error. A parity error occurred on the data portion of the DDI.

11.6 PROGRAMMING HINTS

This section contains information on important programming considerations relevant to users of the DR32 driver described in this chapter.

11.6.1 Command Packet Pre-fetch

The DR32 has the capability of pre-fetching command packets from INPTQ. While executing the command specified in one packet, the DR32 can pre-fetch the next packet, decode it, and be ready to execute the specified command at the first opportunity. When the command is executed depends on which command is specified. For example, if two read device or write device command packets are on INPTQ, the DR32 fetches the first packet, decodes the command, verifies that the transfer is legal, and starts the data transfer. While the transfer is taking place, the DR32 pre-fetches the next read device or write device command packet, decodes it, and verifies the transfer legality. The second transfer begins as soon as the first transfer is completed.

On the other hand, if the two command packets on INPTQ are read device (or write device) and write device control message, in that order, the DR32 pre-fetches the second packet and immediately executes the command, because control messages can be overlapped with data transfers. The DR32 then pre-fetches the next command packet. In an extreme case, the DR32 can send several control messages over the control portion of the DDI while a single data transfer takes place on the data portion of the DDI.

The pre-fetch capability and the overlapping of control and data transfers can cause unexpected results when programming the DR32. For instance, if the application program calls for a data transfer to the far end DR-device followed by notification of the far end DR-device that data is present, the program cannot simply insert a write device command packet and then a write control message command packet onto INPTQ -- the control message may very likely arrive before the data transfer completes.

A better way to synchronize the data transfer with notification of data arrival is to request an interrupt in the interrupt control field of the data transfer command packet. Then, when the data transfer command packet is removed from TERMQ, the application program can insert a write control message command packet onto INPTQ to notify the far end DR-device that the data transfer has completed.

Another consequence of command packet pre-fetching occurs when, for example, two write device command packets are inserted onto INPTQ. While the first data transfer takes place, the second command packet is pre-fetched and decoded. If an unusual event occurs and the application program must send an immediate control message to the far end DR-device, the application program may insert a write device control message packet onto INPTQ. However, this packet is not sent immediately because the second write device command packet has already been pre-fetched; the control message is sent after the second data transfer starts.

If the application program requires the ability to send a control message with minimum delay, use one of the following techniques:

- Insert only one data transfer function onto INPTQ at a time. If this is done, a second transfer function will not be pre-fetched and a control message can be sent at any time.

- Use smaller buffers or a faster data rate to reduce the time necessary to complete a given command packet.
- Issue a $CANCEL system service call followed by another IO$_STARTDATA QIO.

11.6.2 Action Routines

Action routines provide a useful DR32 programming technique. They can be used in application programs written in either assembly language or a high-level language. When a command packet is built, the address of a routine to be executed when the packet is removed from TERMQ is appended to the end of the packet. Then, rather than having to determine what action to perform for a particular packet when it is removed from TERMQ, the specified action routine is called.

11.6.3 Error Checking

Bits 0 through 23 in the second longword of the I/O status block correspond to the same bits in the DR32 status longword (DSL). Although the I/O status block is written only after the QIO function completes, the DSL is stored in every command packet. However, because there is no command packet in which to store a DSL for certain error conditions, for example, FREEQ empty, some errors are reported only in the I/O status block. To check for an error under these conditions, the user should examine the DSL in each packet for success or failure only. Then, if a failure occurs, the specific error can be determined from the I/O status block. The I/O status block should also be checked to verify that the QIO has not completed prior to a wait for the insertion of additional command packets onto TERMQ. In this way, the application program can detect asynchronous errors for which there is no command packet available.

11.6.4 Queue Retry Macro

When an interlocked queue instruction is included in the application program, the code should perform a retry if the queue is locked. However, the code should not execute an indefinite number of retries. Consequently, all retry loops should contain a maximum retry count. The macro programming example provided in Section 11.7 contains a convenient queue retry macro.

11.6.5 Diagnostic Functions

The diagnostic functions listed in Table 11-2 can be used to test the DR32 without the presence of a far end DR-device. For the DR780, the user should perform the following test sequence:

1. Insert a set self-test command packet onto INPTQ.
2. Insert a diagnostic write internal command packet that specifies a 128-byte buffer onto INPTQ. This packet copies 128 bytes from memory into the DR780 internal data silo.

3. Insert a diagnostic read DDI command packet onto INPTQ. This packet transmits the 128 bytes of data from the silo over the DDI and returns it to the silo.

4. Insert a diagnostic read internal command packet that specifies another 128-byte buffer in memory onto INPTQ. This packet copies 128 bytes of data from the silo into memory.

5. Compare the two memory buffers for equality. Note that on the DR780, the diagnostic read internal function destroys the first four bytes in the silo before storing the data in memory. Therefore, compare only the last 124 bytes of the two buffers.

6. Insert a clear self-test command packet onto INPTQ.

11.6.6 The NOP Command Packet

It is often useful to insert a NOP command packet onto INPTQ to test the state of the DDI disable bit (XF$M_PKT_DDIDIS in the DSL). By checking this bit before initiating a data transfer, an application program can determine if the far end DR-device is ready to accept data.

11.6.7 Interrupt Control Field

As described ih Section 11.4.3.6, the interrupt control field determines the conditions under which an interrupt is generated: unconditionally, if TERMQ was empty, or never. There are several general applications of this field:

1. If a program performs five data transfers and requires notification of completion only after all five have completed, the first four command packets should specify no interrupt and the fifth command packet should specify an unconditional interrupt.

2. If a program performs a continuous series of data transfers, for example, each command packet can specify interrupt only if TERMQ was empty. Then, every time an event flag or AST notifies the program that a command packet was inserted onto TERMQ, the program removes and processes all packets on TERMQ until it is empty.

3. Command packets that specify no interrupt should never be mixed with command packets that specify interrupt if TERMQ was empty. If this were done, a command packet that specifies no interrupt could be inserted onto TERMQ followed by a command packet that specifies interrupt if TERMQ was empty. Then the latter packet would not interrupt and the program would never be notified that command packets were inserted onto TERMQ.

11.7 PROGRAMMING EXAMPLES

The program examples in the following two sections use DR32 high-level language procedures and DR32 Queue I/O functions.

11.7.1 DR32 High-level Langauge Program (Program A)

This program is an example of how the DR32 high-level language procedures perform a data transfer from a far end DR-device. The program reads a specified number of data buffers from an undefined far end DR-device, which is assumed to be a data source, into the VAX-11 memory. The number of buffers is controlled by the MAXBUF parameter. The program contains examples of the read data chained function code and DR32 application program synchronization using AST routines and action routines.

```
C  *******************************************************************
C
C                               PROGRAM A
C
C  *******************************************************************


        INCLUDE 'XFDEF.FOR'                     ;DEFINE XF CONSTANTS
        PARAMETER       BUFSIZ = 1024           !SIZE OF EACH BUFFER
        PARAMETER       NUMBUF = 8              !NUMBER OF BUFFERS IN
                                                !RING
        PARAMETER       ILOGSIZ = 4             !SIZE OF INPUT LOG
                                                !ARRAY
        PARAMETER       EFN = 0                 !EVENT FLAG SYNCHRON-
                                                !IZING MAIN LEVEL WITH
                                                !AST ROUTINE

        INTEGER*2       BUFARRAY(BUFSIZ,NUMBUF) !THE RING OF BUFFERS
        INTEGER*2       INDEX                   !REFERS TO BUFFER
                                                !IN BUFARRAY
        INTEGER*2       COUNT                   !COUNTS NUMBER OF
                                                !BUFFERS FILLED
        INTEGER*2       DATART                  !DR32 CLOCK RATE

        INTEGER*4       CONTXT(30)      !CONTEXT ARRAY USED BY SUPPORT
        INTEGER*4       ILOGMSG(ILOGSIZ)!LOG MESSAGES FROM DEVICE
                                        !STORED HERE
        INTEGER*4       STATUS          !RETURNS FROM SUBROUTINES
        INTEGER*4       DEVMSG          !FAR END DR-DEVICE CODE

        EXTERNAL        ASTRTN          !AST ROUTINE
        EXTERNAL        AST$PROCBUF     !ACTION ROUTINE TO HANDLE
                                        !COMPLETION OF READ DATA
                                        !COMMAND PACKET
        EXTERNAL        AST$HALT        !ACTION ROUTINE TO HANDLE
                                        !COMPLETION OF A HALT
                                        !COMMAND PACKET

        COMMON  /MAIN_AST/      CONTXT, INDEX
        COMMON  /MAIN_ACTION/   BUFARRAY, ILOGMSG, COUNT
        EXTERNAL        SS$_NORMAL      !SUCCESS STATUS RETURN

C  ******************************************************************
C
C  THE CALL TO THE SETUP ROUTINE
```

```
C
C  ******************************************************************
        CALL XF$SETUP (CONTXT,BUFARRAY,BUFSIZ*2,NUMBUF,,,ILOGMSG,
        1               ILOGSIZ*4,,STATUS)
        IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))

C
C  PRE-LOAD THE INPUT QUEUE BEFORE STARTING THE DR32 IN ORDER TO AVOID
C  A DELAY IN THE DATA TRANSFER
C

C
  ******************************************************************
C
C  BUILD COMMAND PACKETS
C
C  ******************************************************************

C  BUILD THE COMMAND PACKET THAT WILL INSTRUCT THE FAR END DR-DEVICE
C  TO START SAMPLING.  ARBITRARILY ASSUME THAT THE FAR END DR-DEVICE
C  WILL RECOGNIZE THIS DEVICE MESSAGE.  INSERT THIS PACKET ON THE
C  INPUT QUEUE (INPTQ).
C
        DEVMSG = 25                      !SIGNAL FAR END DR-DEVICE
                                         !"GO"

        CALL XF$PKTBLD (
        1       CONTXT,                  !THE CONTEXT ARRAY
        1       XF$K_PKT_WRTCM,          !WRITE CONTROL MESSAGE
                                         !FUNCTION
        1       ,,                       !NO INDEX OR SIZE
        1       DEVMSG,                  !SIGNAL "GO"
        1       4,                       !SIZE OF DEVMSG IN BYTES
        1       ILOGSIZ*4                !SPACE FOR INPUT LOG
                                         !MESSAGE
        1       XF$K_PKT_UNCOND          !MODES: UNCONDITIONAL
                                         !       INTERRUPT
        1       + XF$K_PKT_CBDM          !     : SEND FUNC AND DEVMSG
        1       + XF$K_PKT_INSTL         !     : INSERT PACKET AT INPTQ
                                         !       TAIL
        1       ''                       !NO ACTION ROUTINE OR ACTPARM
        1       STATUS)
        IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))

C
C  IN A LOOP, BUILD THE COMMAND PACKETS THAT WILL PERFORM THE CHAINED
C  READ TO INITIALLY FILL THE BUFFERS
C

        DO 10   INDEX = 1, NUMBUF        !FOR ALL BUFFERS DO
                CALL XF$PKTBLD(
        1       CONTXT,                  !THE CONTEXT ARRAY
        1       XF$K_PKT_RDCHN,          !READ DATA CHAINED
        1       INDEX,                   !IDENTIFIES BUFFER
        1       ,,,                      !NO SIZE, DEVMSG, OR DEVSIZ
        1       ILOGSIZ*4,               !SPACE FOR INPUT LOG MESSAGE
        1       XF$K_PKT_UNCOND          !MODES: UNCONDITIONAL
                                         !       INTERRUPT
        1       + XF$K_PKT_CB            !     : SEND FUNCTION CODE
        1       + XF$K_PKT_INSTL,        !     : INSERT PACKET AT INPTQ
                                         !       TAIL
        1       AST$PROCBUF,             !ACTION ROUTINE
        1       ,                        !NO ACTPARM
        1       STATUS)
        IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))
10      CONTINUE

C
C  THE INPUT QUEUE IS LOADED
C

C  *******************************************************************
C
C  START THE DR32
C
C  *******************************************************************

        DATART = 0                      !DATA TRANSFER RATE
        COUNT = 0                       !NUMBER OF BUFFERS THAT HAVE
                                        !BEEN FILLED
        CALL SYS$CLREF (%VAL(EFN))      !CLEAR EVENT FLAG BEFORE START

        CALL XF$STARTDEV (CONTXT,'XFA0:',ASTRTN,,,,DATART,STATUS)


        IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))

C
C  FROM THIS POINT, ROUTINES AT THE AST LEVEL ASSUME CONTROL.  WAIT
C  FOR THEM TO SIGNAL COMPLETION OF THE SAMPLING SWEEP.
C
        CALL SYS$WAITFR (%VAL(EFN))

        STOP
        END

C  *******************************************************************
C
C  AST ROUTINES
C
C  *******************************************************************

        SUBROUTINE      ASTRTN (ASTPARM)

        INCLUDE 'XFDEF.FOR/NOLIST'
        INTEGER*2       ASTPARM                 !UNUSED PARAMETER

        INTEGER*4       CONTXT(30)              !CONTEXT ARRAY
        INTEGER*4       STATUS                  !FOR CALL TO XF$GETPKT

        LOGICAL*1       WAITFLG                 !INPUT TO XF$GETPKT
        LOGICAL*1       LOGFLAG                 !INPUT TO XF$GETPKT

        COMMON  /MAIN_AST/      CONTXT, INDEX

        EXTERNAL        SS$_NORMAL

C
C  CALL XF$GETPKT IN A LOOP UNTIL TERMQ IS EMPTY.  XF$GETPKT WILL CALL
C  THE APPROPRIATE ACTION ROUTINE FOR EACH COMMAND PACKET.
C

        WAITFLG = .TRUE.                !DO NOT WAIT FOR EVENT FLAG
        LOGFLAG = .TRUE.                !REQUEST NOTIFICATION IF LOG
                                        !MESSAGE IS IN PACKET

10      CALL XF$GETPKT (CONTXT,WAITFLG,,INDEX,,LOGFLAG,STATUS)
        IF (STATUS .EQ. %LOC(SS$_NORMAL))       !PACKET FROM TERMQ
        1       GOTO 10
        IF (STATUS .EQ. SHR$_QEMPTY)    !TERMQ EMPTY - TRANSFER
        1       GOTO 20                 !STILL IN PROGRESS
        IF (STATUS .EQ. SHR$_HALTED .OR. STATUS .EQ. SHR$_NOCMDMEM)
        1       GOTO 20                 !TRANSFER COMPLETE.  NO MORE
                                        !COMMAND PACKETS. ASTS MAY
                                        !STILL BE DELIVERED

        CALL LIB$STOP (%VAL(STATUS))    !ERROR IN XF$GETPKT

20      RETURN
        END

C  *******************************************************************
C
C  ACTION ROUTINE
C
C  *******************************************************************

        SUBROUTINE      AST$PROCBUF (CONTXT,ACTPARM,DEVFLAG,LOGFLAG,
        1                               FUNC,INDEX,STATUS)

C
C  THIS IS THE ACTION ROUTINE CALLED BY XF$GETPKT WHEN IT REMOVES A
C  COMMAND PACKET FROM TERMQ.  THIS PACKET HAS JUST COMPLETED A READ
C  DATA OPERATION FROM THE BUFFER SPECIFIED BY INDEX.  THE BUFFER IS
C  PROCESSED, AND IF MORE DATA IS REQUIRED (I.E., BUFCOUNT .LE.
C  MAXCOUNT), ANOTHER PACKET IS BUILT.  THE BUFFER IN THIS PACKET IS
C  THEN REFILLED AND THE PACKET IS INSERTED ONTO INPTQ.
C  IF BUFCOUNT .GT. MAXCOUNT, THE SAMPLING SWEEP IS FINISHED AND A
C  HALT PACKET IS INSERTED ONTO INPTQ.
C

        INCLUDE         'XFDEF.FOR/NOLIST'
        PARAMETER       MAXCOUNT = 10   !NUMBER OF BUFFERS IN SWEEP
        PARAMETER       ILOGSIZ = 4     !SIZE OF INPUT LOG MESSAGE
 ARRAY
        PARAMETER       BUFSIZ = 1024   !SIZE OF EACH BUFFER (IN
 WORDS)
        PARAMETER       NUMBUF = 8      !NUMBER OF BUFFERS

        INTEGER*2       INDEX           !REFERS TO A BUFFER IN
 BUFARRAY
        INTEGER*2       FUNC            !FUNCTION CODE FROM PACKET
        INTEGER*2       BUFCOUNT        !COUNTS NUMBER OF BUFFERS
 FILLED
        INTEGER*2       BUFARRAY(BUFSIZ,NUMBUF) !THE ARRAY OF BUFFERS

        INTEGER*4       ACTPARM         !ACTION PARAMETER (NOT USED)
        INTEGER*4       STATUS          !STATUS OF XF$GETPKT (NOT
 USED)
        INTEGER*4       STAT            !STATUS OF CALL TO XF$PKTBLD
        INTEGER*4       CONTXT(30)      !CONTEXT ARRAY USED BY SUPPORT
        INTEGER*4       ILOGMSG(ILOGSIZ)!STORES LOG MESSAGES FROM
 DEVICE

        LOGICAL*1       DEVFLAG         !NOT USED IN THIS EXAMPLE
        LOGICAL*1       LOGFLAG         !SIGNALS LOG MESSAGE PRESENT

        COMMON  /MAIN_ACTION/   BUFARRAY,ILOGMSG,BUFCOUNT

        EXTERNAL        SS$_NORMAL
        EXTERNAL        AST$HALT
```

```
C
C  PROCESS THE BUFFER
C

        DO 10   I = 1, BUFSIZ

C  ********************************************************************
C
C  AT THIS POINT INSERT THE CODE TO PROCESS ELEMENT (I,INDEX) OF
C  BUFARRAY
C
C  ********************************************************************


10      CONTINUE


C  ********************************************************************
C
C  AT THIS POINT INSERT THE CODE TO LOOK AT THE LOG MESSAGE
C
C  ********************************************************************

C
C  IS THIS THE LAST BUFFER IN THE SWEEP?
C

BUFCOUNT = BUFCOUNT + 1
        IF (BUFCOUNT .LT. MAXCOUNT) THEN        !BUILD A PACKET TO
                                         !REFILL THE BUFFER
        CALL FAKE$PKTBLD (               !NEED INTERVENING ROUTINE
        1        CONTXT,                 !THE CONTEXT ARRAY
        1       XF$K_PKT_RDCHN,          !READ DATA CHAINED
        1       INDEX,                   !BUFFER INDEX
        1       ,,,                      !NO SIZE, DEVMSG, OR DEVSIZ
        1       ILOGSIZ*4,               !SPACE FOR LOG MESSAGE
        1       XF$K_PKT_UNCOND          !MODES: UNCONDITIONAL
                                         !       INTERRUPT
        1       + XF$K_PKT_CB            !     : SEND CONTROL BYTE
        1       + XF$K_PKT_INSTL,        !     : INSERT AT TAIL
        1       ,,                       !ACTION GIVEN IN FAKE$PKTBLD
        1       STAT)
        IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))

        ELSE IF (BUFCOUNT .EQ. MAXCOUNT) THEN    !END OF CHAIN
                CALL FAKE$PKTBLD (       !NEED INTERVENING ROUTINE
        1       CONTXT,                  !THE CONTEXT ARRAY
        1       XF$K_PKT_RD,             !READ DATA FUNCTION
        1       INDEX,                   !BUFFER INDEX
        1       ,,,                      !NO SIZE, DEVMSG, OR DEVSIZ
        1       ILOGSIZ*4,               !SPACE FOR LOG MESSAGE
        1       XF$K_PKT_UNCOND          !MODES: UNCONDITIONAL
                                         !       INTERRUPT
        1       + XF$K_PKT_CB            !     : SEND CONTROL BYTE
        1       + XF$K_PKT_INSTL,        !     : INSET AT TAIL
        1       ,,                       !ACTION GIVEN IN FAKE$PKTBLD
        1       STAT)
        IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))

        ELSE                             !BUILD A HALT PACKET
                CALL XF$PKTBLD (
        1       CONTXT,                  !THE CONTEXT ARRAY
        1       XF$K_PKT_HALT,           !ALL DONE
        1       ,,,,                     !DEFAULT VALUES
```

```
     1       ILOGSIZ*1,                  !SPACE FOR INPUT LOG MESSAGE
     1       AST$HALT,                   !ACTION ROUTINE


     1       ,                           !NO ACTPARM
     1       STAT)
     IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))

     END IF

     RETURN
     END

C  ******************************************************************
C
C  PASS ADDRESS OF ACTION ROUTINE TO COMMAND PACKET
C
C  ******************************************************************

     SUBROUTINE      FAKE$PKTBLD(A,B,C,D,E,F,G,H,I,J,K)

C
C  AST$PROCBUF CALLS THIS SUBROUTINE IN ORDER TO PASS THE ADDRESS OF
C  AST$PROCBUF TO XF$PKTBLD.  (AST$PROCBUF CANNOT REFER TO ITSELF
C  WITHIN THE SCOPE OF AST$PROCBUF)
C

     EXTERNAL        AST$PROCBUF

     CALL XF$PKTBLD (A,B,C,D,E,F,G,H,AST$PROCBUF,J,K)

     RETURN
     END

C  ******************************************************************
C
C  HALT ACTION ROUTINE
C
C  ******************************************************************

     SUBROUTINE      AST$HALT (CONTXT,ACTPARM,DEVFLAG,LOGFLAG,
                               FUNC,INDEX,STATUS)

C
C  THIS IS THE ACTION ROUTINE CALLED BY XF$GETPKT WHEN IT REMOVES A
C  HALT PACKET FROM TERMQ.  THIS ROUTINE PRINTS STATUS INFORMATION,
C  CALLS XF$CLEANUP TO PERFORM FINAL HOUSEKEEPING FUNCTIONS, AND SETS
C  THE EVENT FLAG THAT SIGNALS THE TRANSFER IS COMPLETE.
C

     PARAMETER       EFN = 0

     INTEGER*2       FUNC            !NOT USED
     INTEGER*2       INDEX           !NOT USED


     INTEGER*4       ACTPARM         !NOT USED
     INTEGER*4       STATUS          !NOT USED
     INTEGER*4       STAT            !RETURN FROM XF$CLEANUP
     INTEGER*4       CONTXT(30)      !CONTEXT ARRAY USED BY SUPPORT


     LOGICAL*1       DEVFLAG         !NOT USED
     LOGICAL*1       LOGFLAG         !SIGNALS LOG MESSAGE
```

```
        EXTERNAL        SS$_NORMAL      !SUCCESS STATUS RETURN

C
C  PRINT FINAL STATUS
C

        PRINT *, 'FINAL STATUS IN I/O STATUS BLOCK'
        PRINT *, CONTXT(1), CONTXT(2)

C
C  CLEAN UP
C

        CALL XF$CLEANUP (CONTXT,STAT)
        IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))

        CALL SYS$SETEF (%VAL(EFN))

        RETURN
        END
```

11.7.2 DR32 Queue I/O Functions Program (Program B)

This sample program uses QIO functions to send a device message to the far end DR-device and then waits for a message returned in a command packet on FREEQ. The returned message is copied into another command packet and that packet writes a data buffer to the far end DR-device.

```
; ********************************************************************
;
;                               PROGRAM B
;
; ********************************************************************

        .TITLE  DR32 PROGRAMMING EXAMPLE
        .IDENT  /01/

;
; DEFINE SYMBOLS
;

        $XFDEF

;
;
; QRETRY - THIS MACRO EXECUTES AN INTERLOCKED QUEUE INSTRUCTION AND
;          RETRIES THE INSTRUCTION UP TO 25 TIMES IF THE QUEUE IS
;          LOCKED.
;
; INPUTS:
;
;       OPCODE = OPCODE NAME: INSQHI,INSQTI,REMQHI,REMQTI
;       OPERAND1 = FIRST OPERAND FOR OPCODE
;       OPERAND2 = SECOND OPERAND FOR OPCODE
;       SUCCESS = LABEL TO BRANCH TO IF OPERATION SUCCEEDS
;       ERROR = LABEL TO BRANCH TO IF OPERATION FAILS
;
; OUTPUTS:
;
;       R0 = DESTROYED
;
```

```
;       C-BIT = CLEAR IF OPERATION SUCCEEDED
;               SET IF OPERATION FAILED - QUEUE LOCKED
;               (MUST BE CHECKED BEFORE V-BIT OR Z-BIT)
;
;       REMQTI OR REMQHI:
;
;               V-BIT = CLEAR IF AN ENTRY REMOVED FROM QUEUE; SET
;                       IF NO ENTRY REMOVED FROM QUEUE.
;
;       INSQTI OR INSQHI:
;
;               Z-BIT = CLEAR IF ENTRY IS NOT FIRST IN QUEUE; SET
;                       IF ENTRY IS FIRST IN QUEUE.
;
;
        .MACRO  QRETRY  OPCODE,OPERAND1,OPERAND2,SUCCESS,ERROR,?LOOP,
                        ?OK
        CLRL    R0
LOOP:
        OPCODE  OPERAND1,OPERAND2
        .IF     NB      SUCCESS
        BCC     SUCCESS
        .IFF
        BCC     OK
        .ENDC
        AOBLSS  #25,R0,LOOP
        .IF     NB      ERROR
        BRW     ERROR
        .ENDC
OK:
        .ENDM   QRETRY
;
; ALLOCATE STORAGE FOR DATA STRUCTURES
;

        .PSECT  DATA,QUAD
CMDBLK:                                 ; COMMAND BLOCK

INPTQ:  .BLKQ   1                       ; INPUT QUEUE
TERMQ:  .BLKQ   1                       ; TERMINATION QUEUE
FREEQ:  .BLKQ   1                       ; FREE QUEUE

MSGPKT:                                 ; THIS PACKET SENDS A 12 BYTE
                                        ; DEVICE MESSAGE
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   12                      ; LENGTH OF DEVICE MESSAGE
        .BYTE   0                       ; LENGTH OF LOG AREA
        .BYTE   XF$K_PKT_WRTCM          ; COMMAND  = WRITE CONTROL
                                        ; MESSAGE
        .BYTE   XF$K_PKT_NOINT@-        ; PACKET CONTROL = NO
                                        ; INTERRUPT
                XF$V_PKT_INTCTL
        .BLKL   1                       ; BYTE COUNT
        .BLKL   1                       ; BUFFER ADDRESS
        .BLKL   2                       ; RESIDUAL MEMORY AND DDI BYTE
                                        ; COUNTS
        .BLKL   1                       ; DR32 STATUS LONGWORD
        .LONG   11111,22222,33333       ; DEVICE MESSAGE
        .LONG   0                       ; EXTEND DEVICE MESSAGE TO
                                        ; QUADWORD LENGTH

        .ALIGN  QUAD
WRTPKT:                                 ; THIS PACKET DOES A WRITE
                                        ; DEVICE
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   4                       ; LENGTH OF DEVICE MESSAGE
        .BYTE   0                       ; LENGTH OF LOG AREA
        .BYTE   XF$K_PKT_WRT            ; COMMAND = WRITE
        .BYTE   <XF$K_PKT_CBDMBC@-      ; PACKET CONTROL = SEND
                                        ; COMMAND BYTE,
                XF$V_PKT_CISEL>!-       ; DEVICE MESSAGE, AND BYTE
                                        ; COUNT
                <XF$K_PKT_NOINT@-       ; AND NO INTERRUPT
                XF$V_PKT_INTCTL>
        .LONG   1000                    ; BYTE COUNT
        .LONG   WRTBFR                  ; BUFFER ADDRESS
        .BLKL   2                       ; RESIDUAL MEMORY AND DDI BYTE
                                        ; COUNTS
        .BLKL   1                       ; DR32 STATUS LONGWORD
WDVMSG: .BLKQ   1                       ; SPACE FOR DEVICE MESSAGE

        .ALIGN  QUAD


HLTPKT:                                 ; THIS PACKET HALTS THE DR32
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   0,0,XF$K_PKT_HALT,0     ; COMMAND = HALT
        ,BLKL   5                       ; UNUSED FIELDS IN THIS PACKET

        .ALIGN QUAD
FREPKT:                                 ; PACKET FOR FREE QUEUE
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   4,0,0,0                 ; LENGTH OF DEVICE MESSAGE
                                        ; FIELD
        .BLKL   4                       ; UNUSED FIELDS IN THIS PACKET
        .BLKL   1                       ; DR32 STATUS LONGWORD
        .BLKQ   1                       ; SPACE FOR DEVICE MESSAGE

CMDBLKSIZ=.-CMDBLK


BFRBLK:                                 ; BUFFER BLOCK



WRTBFR: .BLKB   1000

BFRBLKSIZ=.-BFRBLK

CMDTBL: .LONG   CMDBLKSIZ               ; COMMAND BLOCK SIZE
        .LONG   CMDBLK                  ; COMMAND BLOCK ADDRESS
        .LONG   BFRBLKSIZ               ; BUFFER BLOCK SIZE
        .LONG   BFRBLK                  ; BUFFER BLOCK ADDRESS
        .LONG   PKTAST                  ; PACKET AST ADDRESS
        .LONG   0                       ; PACKET AST PARAMETER
        .BYTE   236,XF$M_CMT_SETRTE,0,0 ; DATA RATE (2.0 MBYTES/SEC)
        .LONG   GOBITADR                ; ADDRESS TO STORE THE GO
                                        ; BIT ADDRESS

GOBITADR:
        .BLKL   1

XFIOSB: .BLKL   2                       ; I/O STATUS BLOCK

XFNAMEDSC:
        .LONG   XFNAMESIZ               ; NAME DESCRIPTOR
        .LONG   XFNAME

XFCHAN: .BLKW   1                       ; CHANNEL NUMBER
```

```
XFNAME: .ASCII  /XFA0/
XFNAMESIZE=.-XFNAME

; *********************************************************************
;
; PROGRAM STARTING POINT
;
; *********************************************************************

        .PSECT  CODE,NOWRT

        .ENTRY  DREXAMPLE,M<R2,R3>

        $ASSIGN_S DEVNAM = XFNAMEDSC,-  ; ASSIGN A CHANNEL TO DR32
                  CHAN = XFCHAN
        BLBS    R0,10$                  ; SUCCESSFUL ASSIGN
        BRW     ERROR

10$:    MOVAB   CMDBLK,R2
        CLRQ    (R2)+                   ; INITIALIZE INPTQ
        CLRQ    (R2)+                   ; INITIALIZE TERMQ
        CLRQ    (R2)                    ; INITIALIZE FREEQ

;
; INSERT COMMAND PACKET ONTO FREEQ FOR RETURN MESSAGE
;

        QRETRY  ERROR=BADQUEUE,-
        INSQTI  FREPKT,FREEQ

;
; START DEVICE
;

        $QIO_S  FUNC = #IO$_STARTDATA,-


                CHAN = XFCHAN,-
                IOSB = XFIOSB,-
                EFN = #1,-
                P1 = CMDTBL,-
                P2 = #XF$K_CMT_LENGTH
        BLBC    R0,ERROR

;
; SEND MESSAGE TO FAR END DR-DEVICE
;

        QRETRY  ERROR=BADQUEUE,-
        INSQTI  MSGPKT,INPTQ
        MOVL    #1,@GOBITADR            ; SET GO BIT
        $WAITFR_S #1                    ; WAIT UNTIL QIO COMPLETES

;
; CHECK FOR SUCCESSFUL COMPLETION
;

        MOVZWL  XFIOSB,R0
        BEQL    BADQUEUE                ; I/O NOT DONE YET - BAD QUEUE


                                        ; ERROR IN AST ROUTINE
        BLBC    R0,ERROR                ; ERROR
        RET                             ; SUCCESSFUL COMPLETION
```

```
BADQUEUE:
        MOVZWL  #SS$_BADQUEUEHDR,R0

;
; AN ERROR HAS OCCURRED.  NORMALLY, THE USER MIGHT PERFORM MORE
; EXTENSIVE ERROR CHECKING AT THIS POINT.  IN PARTICULAR, IF THE ERROR
; IS SS$_CTRLERR, SS$_DEVREQERR, OR SS$_PARITY, THE SECOND LONGWORD
; OF THE I/O STATUS BLOCK CAN PROVIDE ADDITIONAL INFORMATION.  IN THIS
; EXAMPLE, THE PROGRAM EXITS WITH THE ERROR STATUS IN R0.
;


ERROR:  RET


;
; COMMAND PACKET AST ROUTINE
;

PKTAST: .WORD   0
NXTPKT: QRETRY  ERROR=70$,-                ; GET NEXT PACKET FROM QUEUE
        REMQHI  TERMQ,R1
        BVC     10$                        ; PACKET OBTAINED FROM QUEUE
        RET                                ; QUEUE IS EMPTY
10$:    BLBC    XF$L_PKT_DSL(R1),50$       ; RETURN IF PACKET ERROR
        BBC     #XF$V_PKT_FREQPK,-         ; RETURN IF PACKET NOT FROM
                XF$L_PKT_DSL(R1),50$       ; FREEQ

;
; COMMAND PACKET OBTAINED FROM FREEQ.  COPY DEVICE MESSAGE AND QUEUE
; WRITE PACKET.
;

        MOVL    XF$B_PKT_DEVMSG(R1),WDVMSG
        QRETRY  ERROR=70$,-
        INSQTI  WRTPKT,INPTQ
        QRETRY  ERROR=70$,-
        INSQTI  HLTPKT,INPTQ
        MOVL    #1,@GOBITADR               ; SET GO BIT
50$:    RET

;
; BAD QUEUE ERROR IN AST ROUTINE - WAKE UP MAIN LEVEL.  QIO MAY
; OR MAY NOT HAVE COMPLETED.
;

70$:    $SETEF_S  #1                       ; WAKE UP MAIN LEVEL
        RET

        .END    DREXAMPLE
```

CHAPTER 12

DUP11 INTERFACE DRIVER

This chapter describes the use of the DUP11 Device Interface driver. The driver is category C software, which is not supported. The DUP11 is the lowest-level user interface to the VAX/VMS 2780/3780 Protocol Emulator. (The user can also access the 2780/3780 through the command language interface and the record-oriented interface. See the VAX/VMS 2780/3780 Protocol Emulator User's Guide.)

12.1 SUPPORTED DEVICE

The DUP11 is a single line, program-controlled communications device that interfaces a VAX-11 processor to a serial, synchronous communications line. Data transmission occurs at a maximum speed of 9600 baud. Although the DUP11 functions in either full- or half-duplex mode, the DUP11 driver operates logically only in half-duplex mode; only one I/O request is processed at any given time but many may be queued.

The DUP11 driver transfers output data from the VAX/VMS system to the DUP11. The DUP11 then shifts the data onto the communications line. Input data from the communications line modem is shifted into the DUP11 where it is made available to the DUP11 driver on an interrupt basis.

12.1.1 Driver Operating Modes

The device driver functions in two operating modes: binary synchronous communications (BSC) mode and binary mode. BSC mode operations are described in Appendix C of the VAX/VMS 2780/3780 Protocol Emulator User's Guide. The preface of the same manual also provides a list of related documents.

In BSC mode, the driver observes standard point-to-point BSC protocol in send and receive operations. In binary mode, the driver does not observe any protocol; the only operation performed on the data is the insertion or deletion of PAD and SYN characters. An operation is completed when the buffer count reaches zero or the I/O is cancelled.

Function modifiers, which are included in all read and write requests to the driver, define the operating mode for each I/O operation.

If the only reason for not using the record-oriented interface is the blocksize restriction (the application is compatible with all other 2780/3780 communications protocols), the DUP11 driver should be used primarily in BSC mode rather than binary mode. Binary mode is used only if the user requires direct control of some aspect of the communications protocol handled by the driver when in BSC mode. All line protocol messages, for example, bids and sending EOTs, must be transmitted in binary mode.

12.1.1.1 **BSC Mode** - If the IO$M_PTPBSC function modifier is included in a read or write QIO request, data is read or written in BSC mode. The DUP11 driver performs the following operations:

1. Inserts in the output data, and removes from the input data, BSC data-link control characters, for example, STX and ITB.

2. Checks input message blocks for transmission errors. Adds cyclic redundancy check (CRC) characters to output message blocks to support error checking by the communications processor in the remote system.

3. Manages line protocols, for example, ACK, NAK, and ENQ responses, that determine whether a message block must be retransmitted because of transmission errors.

4. Inserts in the output data, and removes from the input data, DLE information in transparent mode.

The DUP11 driver does not modify the input or output data in any way. All necessary processing, for example, data translation and space compression or expansion, must be included in the user program. The user program builds the message block to be transmitted into a single buffer. This buffer must start with a 2-byte count that includes all data up to the point where a CRC will be placed, and end with a 2-byte count field equal to -1. The driver inserts an ITB character in front of internal CRC characters.

Figures 12-1 through 12-5 illustrate how the DUP11 driver reformats user-formatted output message blocks into standard 2780/3780 message blocks. The driver deblocks input messages in the reverse order of that shown in these figures.

All COUNT and CRC fields in these examples are two bytes long. Each record count results in the generation of a CRC character. An ITB character precedes all internal CRC characters. An ETB precedes the last CRC in a block unless the IO$M_LASTBLOCK function modifier is specified. In that case, an ETX precedes the CRC. If in transparency mode (specified by IO$_SETMODE), all data-link control characters are preceded by a DLE character and all DLE characters in the data buffers are changed to DLE DLE. Also, the control character sequence of SYN, SYN, DLE, STX is inserted between records within the message block.

Message blocks transmitted by the DUP11 driver include a prefix of SYN characters (as specified by the set mode QIO) and a suffix of a PAD character (hexadecimal FF).

Figure 12-1 shows the format of user-built message buffers that simulate 3780 processing. The user must pass the buffers to the device driver by issuing QIO requests that include the IO$M_PTPBSC function modifier.

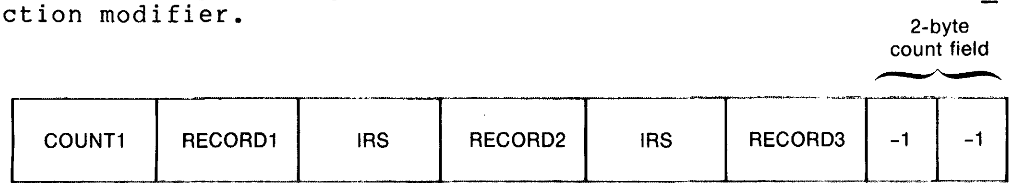


Figure 12-1 3780 Message Block Example

The DUP11 driver transmits the message block after modifying the format, as shown in Figure 12-2. The driver does not modify the data records in the two buffers; they are identical.

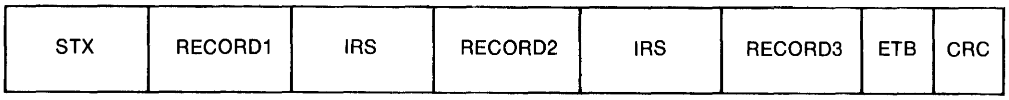


Figure 12-2 3780 Message Block Example (Modified)

To simulate 2780 processing in nontransparent mode, the user builds message buffers in the format shown in Figure 12-3. The user must include the IO$M_PTPBSC function modifier in the QIO requests that pass the buffers to the DUP11 driver.

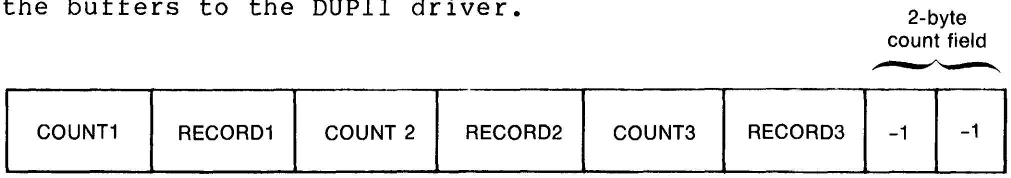


Figure 12-3 Nontransparent 2780 Message Block Example

The DUP11 driver transmits the message block after modifying the format, as shown in Figure 12-4.

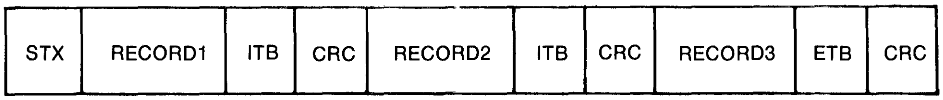


Figure 12-4 Nontransparent 2780 Message Block Example (Modified)

To simulate 2780 processing in transparent mode, the user must specify the transparency modifier in a set mode QIO request, build message buffers in the format shown in Figure 12-3, and include the IO$M_PTPBSC function modifier in the write QIO requests that pass the buffers to the DUP11 driver. The driver transmits the message block after modifying the format, as shown in Figure 12-5. The driver adds a duplicate DLE character to any DLE character encountered in the data records.

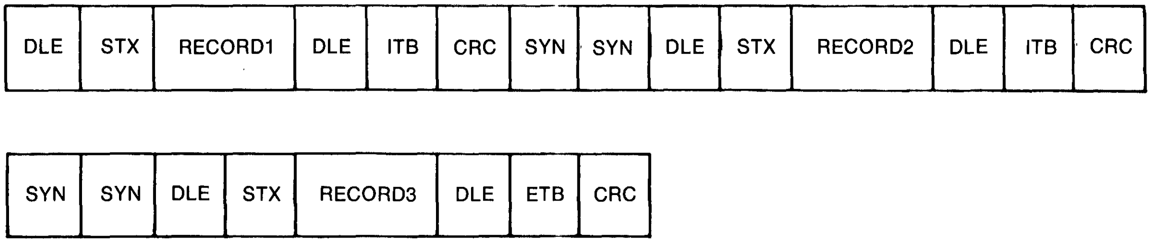


Figure 12-5 Transparent 2780 Message Block Example (Modified)

12.1.1.2 **Binary Mode** - If the IO$M_SRRUNOUT function modifier is included in a read or write request, data is read or written in binary mode. In binary mode, the DUP11 driver performs no processing operations on the user-supplied message block buffer. Except for the insertion in output message blocks, and deletion from input message blocks, of leading SYN and trailing PAD characters, data passes through the DUP11 driver as unprocessed, binary information. The user program directly controls all data transmitted or received by the driver. QIO requests in the user program provide all necessary communications to the remote system. The user program must perform the following functions:

1. Explicitly issue all protocol messages, for example, ACK, NAK, and ENQ responses, to the DUP11 driver.

2. Perform all validity checking calculations and comparisons.

3. Handle the insertion and removal of any message-framing and inter-record control characters in the message blocks.

4. Repeat write QIO requests until the operation is successful or the program's error threshold is reached.

12.2 DEVICE INFORMATION

Users can obtain information on DUP11 characteristics by using the $GETCHN and $GETDEV system services (see Section 1.10). The DUP11-specific information is returned in the first three longwords of a user-specified buffer, shown in Figure 12-6 (Figure 1-9 shows the entire buffer).

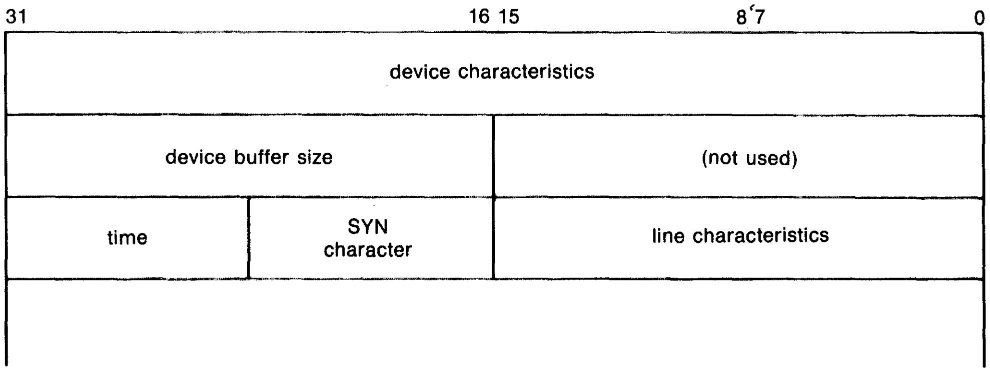


Figure 12-6 DUP11 Information

The first longword contains device-independent information. The second and third longwords contain device-dependent data.

Table 12-1 lists the device-independent characteristics returned in the first longword.

Table 12-1
Device-Independent Characteristics

Dynamic Bits[1] (Conditionally Set)	Meaning
XJ$M_CHA_FDX	Full-duplex line
XJ$M_CHA_XPR	Transparency mode
XJ$M_CHA_DSR	Data set ready
Static Bits[2] (always Set)	
DEV$M_AVL	Device available
DEV$M_IDV	Input device
DEV$M_ODV	Output device

[1] Defined by the $XJDEF macro.

[2] Defined by the $DEVDEF macro.

The second longword contains the device buffer size (default is 520 bytes). The third longword contains the line characteristics, the SYN character, and the time, in seconds, to wait for clear to send (CTS). The SYN character is that character selected to precede all message blocks transmitted by the DUP11 driver. The line characteristics returned in the third longword are:

- XJ$M_CHA_DSC -- Sense state of data terminal ready (DTR) signal line. Meaningful only to IO$_SENSEMODE.
- XJ$M_CHA_FDX -- full duplex mode. Do not drop request to send (RTS) after each segment is transmitted.
- XJ$M_CHA_XPR -- transparent mode. Used only when IO$M_PTPBSC is specified with a write QIO function.

The device buffer size and the third longword contents are established by IO$_SETMODE (see Section 12.3.3).

12.3 DUP11 FUNCTION CODES

The DUP11 can perform logical and physical I/O operations. The basic I/O functions are read, write, set mode, and sense mode. Table 12-2 lists these functions and their function codes. The following paragraphs describe these functions in greater detail.

Table 12-2
DUP11 I/O Functions

Function Code and Arguments	Type[1]	Function Modifiers	Function
IO$_READLBLK P1,P2	L	IO$M_SRRUNOUT IO$M_PTPBSC	Read logical block
IO$_READPBLK P1,P2	P	IO$M_SRRUNOUT IO$M_PTPBSC	Read physical block
IO$_WRITELBLK P1,P2	L	IO$M_SRRUNOUT IO$M_PTPBSC IO$M_LASTBLOCK[2]	Write logical block
IO$_WRITEPBLK P1,P2	P	IO$M_SRRUNOUT IO$M_PTPBSC IO$M_LASTBLOCK[2]	Write physical block
IO$_SETMODE P1	L	IO$M_STARTUP IO$M_NODSRWAIT[3] IO$M_SHUTDOWN	Set line state or line parameters
IO$_SENSEMODE	L		Sense line state; return status

1. L = logical, P = physical

2. Use only with IO$M_PTPBSC

3. Use only with IO$M_STARTUP

12.3.1 Read

Read functions provide for the transfer of data from the DUP11 into the user process's virtual memory address space. VAX/VMS provides two function codes:

- IO$_READLBLK -- read logical block
- IO$_READPBLK -- read physical block

The read function codes take two device/function-dependent arguments:

- P1 = the starting virtual address of the buffer that is to receive data
- P2 = the size of the data buffer in bytes

The read QIO functions can take two function modifiers:

- IO$M_SRRUNOUT -- read data in binary format (see Section 12.1.1.2).
- IO$M_PTPBSC -- read data in BSC mode (see Section 12.1.1.1).

12.3.2 Write

Write functions provide for the transfer of data to the DUP11 from the user process's virtual memory address space. VAX/VMS provides two function codes:

- IO$_WRITELBLK -- write logical block
- IO$_WRITEPBLK -- write physical block

The write function codes take two device/function-dependent arguments:

- P1 = the starting virtual address of the buffer that is to send data to the DUP11
- P2 = the size of the data buffer in bytes

The write QIO functions can take three function modifiers:

- IO$M_SRRUNOUT -- write data in binary format (see Section 12.1.1.2).
- IO$M_PTPBSC -- write data in BSC mode (see Section 12.1.1.1).
- IO$M_LASTBLOCK -- terminate the data block with an ETX character. This function modifier can be used only in conjunction with IO$M_PTPBSC.

12.3.3 Set Mode

The set mode function is used to change the state of the communication line or the parameters that control the line. VAX/VMS provides one function code:

IO$_SETMODE

This function code takes the following device/function-dependent argument:

P1 = points to a quadword buffer block that contains the new communication line parameters

Figure 12-7 shows the format of the P1 buffer.

31 24	23 16	15 0
blocksize		not used
time	SYN character	line characteristics

Figure 12-7 Set Mode P1 Buffer

In the first longword, blocksize is the largest buffer expected. This parameter is included in the buffer block only when an IO$_READLBLK request includes the IO$M_PTPBSC function modifier.

The first word of the second longword specifies the following line characteristics:

- XJ$M_CHA_DSC -- sense state of data terminal ready (DTR) signal line. Meaningful only to IO$_SENSEMODE.

- XJ$M_CHA_FDX -- full duplex mode. Do not drop request to send (RTS) after each segment is transmitted.

- XJ$M_CHA_XPR -- transparent mode. Used only when IO$M_PTPBSC is specified with a write QIO function.

The third byte of the second longword is the SYN character that precedes all message blocks transmitted by the DUP11 driver. The fourth byte specifies the time, in seconds, to wait for clear to send (CTS). This parameter is included in the buffer block only when a read or write request specifies the IO$M_SRRUNOUT function modifier.

The Set Mode function can take three function modifiers:

- IO$M_STARTUP -- enable the communication line (assert data terminal ready (DTR) and wait for data set ready (DSR).

- IO$M_NODSRWAIT -- complete this function without regard to the state of DSR. Used only in conjunction with the IO$M_STARTUP function modifier.

- IO$M_SHUTDOWN -- disable the communication line (disable DTR)

12.3.4 Sense Mode

The sense mode function senses the current state of the communication line and returns the line characteristics and status in the I/O status block (see Figure 12-9). VAX/VMS provides one function code:

IO$_SENSEMODE

12.4 I/O STATUS BLOCK

Figure 12-8 shows the I/O status block for all DUP11 QIO functions except sense mode. Figure 12-9 shows the I/O status block for the sense mode function. Table 12-3 lists the status returns for all functions.

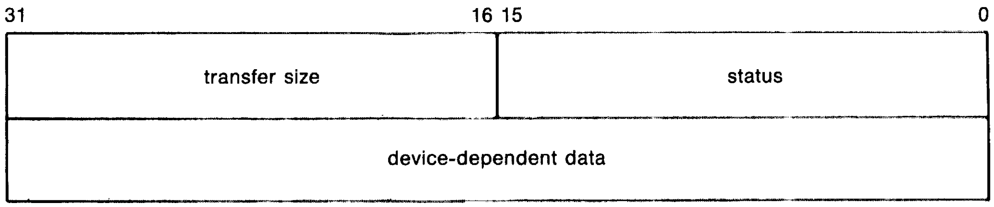


Figure 12-8 IOSB Content

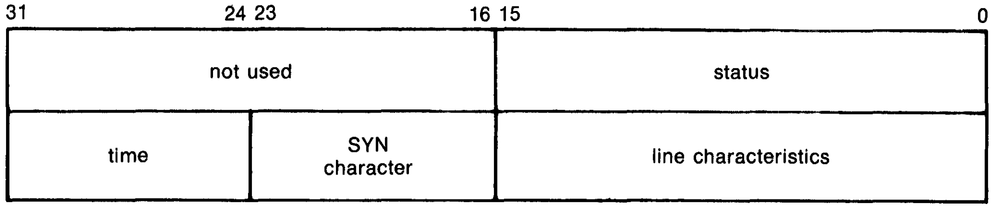


Figure 12-9 IOSB Content - Sense Mode

Table 12-3
DUP11 Status Returns

Status	Meaning
SS$_ABORT	Request aborted. A request in progress was aborted by the $CANCEL system service.
SS$_ACCVIO	Buffer access violation. An attempt was made to read from or write to a location in memory that is protected against the current mode.
SS$_EXQUOTA	Buffered I/O quota exceeded. A request cannot be queued because the buffered I/O quota was exceeded.
SS$_INSFMEM	Insufficient dynamic memory to initiate a data transfer request.
SS$_NORMAL	QIO transfer request completed successfully; the specified data was transferred.

In Figure 12-8, the second word of the first longword contains the size of the transfer in bytes. For transmit (write) operations, the transfer size is the value specified in the P2 argument. For read (receive) operations, transfer size is the amount of data received as the result of the read request. Table 12-4 lists the device-dependent data returned in the second longword.

Table 12-4
Device-Dependent Status Returns

Value	Meaning
XJ$M_BADCHAIN	A RECORD LIST was incorrectly found in a BSC (IO$M_PTPBSC) write request. This is a fatal error condition.
XJ$M_CONACK	A BSC (IO$M_PTPBSC) write request was completed with a conversational ACK character. The data block is considered acknowledged. However, the data received with the ACK character is lost.

(continued on next page)

Table 12-4 (Cont.)
Device-Dependent Status Returns

Value	Meaning
XJ$M_DATACK	Retry threshold exceeded. This is a fatal error condition.
XJ$M_DISCON	BSC disconnect sequence received, that is, DLE, EOT. This is a fatal error condition.
XJ$M_EOT	EOT received. This is a fatal error condition.
XJ$M_EXTEND	A BSC (IO$M_PTPBSC) read request completed successfully. The read data included a block that ended with an EXT character.
XJ$M_ILLMOD	Illegal QIO function modifier detected. This is a fatal error condition.
XJ$M_NODSR	Request aborted because of DSR loss. This is a fatal error condition.
XJ$M_PIPE_MARK	A BSC (IO$M_PTPBSC) transfer aborted because of a previous failure. This is a fatal error condition.
XJ$M_RVI	A BSC (IO$M_PTPBSC) write request completed with a received RVI.
XJ$M_TRABINTMO	A timeout occurred during a binary (IO$M_SRRUNOUT) data transfer. This is a fatal error condition.
XJ$M_XPR	A BSC (IO$M_PTPBSC) read request was satisfied with a transparent block. The received information was transmitted (written) in transparency mode.

In Figure 12-9, the first longword contains the current status of the communication line. Table 12-3 lists the status return values and their meaning.

The first word of the second longword returns one or more of the following line characteristics:

- XJ$M_CHA_DSC -- state of DTR line
- XJ$M_CHA_FDX -- full duplex mode. (Do not drop RTS after each segment tranmitted.)
- XJ$M_CHA_XPR -- transparent mode. Used only when IO$M_PTPBSC is specified with a write QIO function.

The third byte of the second longword is the SYN character selected to precede all message blocks transmitted by the DUP11 driver. The fourth byte specifies the time, in seconds, to wait for clear to send (CTS). This parameter is included in the buffer block only when the IO$M_SRRUNOUT function modifier is specified in a read or write request.

APPENDIX A

I/O FUNCTION CODES

This appendix lists the function codes and function modifiers defined in the $IODEF macro. The arguments for these functions are also listed.

A.1 TERMINAL DRIVER

Function	Arguments	Modifier
IO$_READVBLK	P1 - buffer address	IO$M_NOECHO
IO$_READLBLK	P2 - buffer size	IO$M_CVTLOW[3]
IO$_READPBLK	P3 - timeout	IO$M_NOFILTR[3]
IO$_READPROMPT	P4 - read terminator	IO$M_TIMED
IO$_TTYREADALL	block address	IO$M_PURGE
IO$_TTYREADPALL	P5 - prompt string	IO$M_DSABLMBX
	buffer address[1]	IO$M_TRMNOECHO
	P6 - prompt string	IO$M_REFRESH
	buffer size[1]	
IO$_WRITEVBLK	P1 - buffer address	IO$M_CANCTRLO
IO$_WRITELBLK	P2 - buffer size	IO$M_ENABLMBX
IO$_WRITEPBLK	P3 - (ignored)	IO$M_NOFORMAT
	P4 - carriage control	IO$M_REFRESH
	specifier[2]	
IO$_SETMODE	P1 - characteristics	
IO$_SETCHAR	buffer address	
	P3 - speed specifier	
	P4 - fill specifier	
	P5 - parity flags	
IO$_SETMODE!IO$M_HANGUP	(none)	
IO$_SETCHAR!IO$M_HANGUP		
IO$_SETMODE!IO$M_CTRLCAST	P1 - AST service routine address	
IO$_SETMODE!IO$M_CTRLYAST	P2 - AST parameter	
IO$_SETCHAR!IO$M_CTRLCAST	P3 - access mode to deliver AST	
IO$_SETCHAR!IO$M_CTRLYAST		
IO$_SENSEMODE	P1 - Characteristics	IO$M_TYPEAHDCNT
IO$_SENSECHAR	buffer address	

1. Only for IO$_READPROMPT and IO$_TTYREADPALL

2. Only for IO$_WRITELBLK and IO$_WRITEVBLK

3. Only for IO$_READLBLK,IO$_READVBLK, and IO$_READPROMPT

A.2 DISK DRIVERS

Functions	Arguments	Modifiers
IO$_READVBLK	R1 - buffer address	IO$M_DATACHECK
IO$_READLBLK	P2 - byte count	IO$M_INHRETRY
IO$_READPBLK	P3 - disk address	IO$M_INHSEEK [1]
IO$_WRITEVBLK		
IO$_WRITELBLK		
IO$_WRITEPBLK		
IO$_WRITECHECK	P1 - buffer address	
	P2 - byte count	
	P3 - disk address	
IO$_SETMODE	P1 - characteristic buffer	
IO$_SETCHAR	address	
IO$_SENSECHAR	(none)	
IO$_SENSEMODE		
IO$_PACKACK		
IO$_MOUNT		
IO$_SEARCH	P1 - read/write head position	
IO$_SEEK	P1 - seek to specified cylinder	
IO$_CREATE	P1 - FIB descriptor address	IO$M_CREATE [2]
IO$_ACCESS	P2 - file name string	IO$M_ACCESS [2]
IO$_DEACCESS	address	IO$M_DELETE [3]
IO$_MODIFY	P3 - result string length	IO$M_DMOUNT [4]
IO$_DELETE	address	
IO$_ACPCONTROL	P4 - result string descriptor	
	address	
	P5 - attribute list address	

1. Only for IO$_READPBLK and IO$_WRITEPBLK

2. Only for IO$_CREATE and IO$_ACCESS

3. Only for IO$_CREATE and IO$_DELETE

4. Only for IO$_ACPCONTROL

A.3 MAGNETIC TAPE DRIVERS

Functions	Arguments	Modifiers
IO$_READVBLK	P1 - buffer address	IO$M_DATACHECK [1]
IO$_READLBLK	P2 - byte count	IO$M_INHRETRY
IO$_READPBLK		IO$M_REVERSE [2]
IO$_WRITEVBLK		IO$M_INHEXTGAP [3]
IO$_WRITELBLK		
IO$_WRITEPBLK		

1. Not for TS11

2. Only for read functions

3. Only for write functions

I/O FUNCTION CODES

Functions	Arguments	Modifiers
IO$_SETMODE IO$_SETCHAR	P1 - characteristics buffer address	
IO$_CREATE IO$_ACCESS IO$_DEACCESS IO$_MODIFY IO$_ACPCONTROL	P1 - FIB descriptor address P2 - file name string address P3 - result string length address P4 - result string descriptor address P5 - attribute list address	IO$M_CREATE[1] IO$M_ACCESS[1] IO$M_DMOUNT[2]
IO$_SKIPFILE	P1 - skip n tape marks	IO$M_INHRETRY
IO$_SKIPRECORD	P1 - skip n records	IO$M_INHRETRY
IO$_MOUNT	(none)	
IO$_REWIND IO$_REWINDOFF	(none)	IO$M_INHRETRY IO$M_NOWAIT
IO$_WRITEOF	(none)	IO$M_INHEXTGAP IO$M_INHRETRY
IO$_SENSEMODE	(none)	IO$M_INHRETRY

1. Only for IO$_CREATE and IO$_ACCESS

2. Only for IO$_ACPCONTROL

A.4 LINE PRINTER DRIVER

Functions	Arguments	Modifiers
IO$_WRITEVBLK IO$_WRITELBLK IO$_WRITEPBLK	P1 - buffer address P2 - buffer size P3 - (ignored) P4 - carriage control specifier[1]	(none)
IO$_SETMODE IO$_SETCHAR	P1 - characteristics buffer address	(none)

[1] Only for IO$_WRITEVBLK and IO$_WRITELBLK

A.5 CARD READER DRIVER

Functions	Arguments	Modifiers
IO$_READLBLK IO$_READVBLK IO$_READPBLK	P1 - buffer address P2 - byte count	IO$M_BINARY IO$M_PACKED
IO$_SETMODE IO$_SETCHAR	P1 - characteristics buffer address	(none)
IO$_SENSEMODE	(none)	

A.6 MAILBOX DRIVER

Functions	Arguments	Modifiers
IO$_READVBLK	P1 - buffer address	IO$M_NOW
IO$_READLBLK	P2 - buffer size	
IO$_READPBLK		
IO$_WRITEVBLK		
IO$_WRITELBLK		
IO$_WRITEPBLK		
IO$_WRITEOF	(none)	
IO$_SETMODE!IO$M_READATTN	P1 - AST address	
IO$_SETMODE!IO$M_WRTATTN	P2 - AST parameter	
	P3 - Access mode	

A.7 DMC11 DRIVER

Functions	Arguments	Modifiers
IO$_READLBLK	P1 - buffer address	IO$M_DSABLMBX[1]
IO$_READPBLK	P2 - message size	IO$M_NOW[1]
IO$_READVBLK		IO$M_ENABLMBX[2]
IO$_WRITELBLK		
IO$_WRITEPBLK		
IO$_WRITEVBLK		
IO$_SETMODE	P1 - characteristics	
IO$_SETCHAR	buffer address	
IO$_SETMODE!IO$M_ATTNAST	P1 - AST service	
IO$_SETCHAR!IO$M_ATTNAST	routine address	
	P2 - (ignored)	
	P3 - AST access mode	
IO$_SETMODE!IO$M_SHUTDOWN	P1 - characteristics	
IO$_SETCHAR!IO$M_SHUTDOWN	block address	
IO$_SETMODE!IO$M_STARTUP	P1 - characteristics	
IO$_SETCHAR!IO$M_STARTUP	block address	
	P2 - (ignored)	
	P3 - receive message blocks	

1. Only for read functions

2. Only for IO$_WRITELBLK and IO$_WRITEPBLK

A.8 ACP INTERFACE DRIVER

Functions	Arguments	Modifiers
IO$_CREATE IO$_ACCESS IO$_DEACCESS IO$_MODIFY IO$_DELETE IO$_ACPCONTROL	P1 - FIB descriptor address P2 - file name string address P3 - result string length address P4 - result string descriptor address P5 - attribute list address	IO$M_CREATE[1] IO$M_ACCESS[1] IO$M_DELETE[2] IO$M_DMOUNT[3]
IO$_MOUNT	(none)	

1. Only for IO$_CREATE and IO$_ACCESS

2. Only for IO$_CREATE and IO$_DELETE

3. Only for IO$_ACPCONTROL

A.9 LPA11-K DRIVER

QIO Functions	Arguments	Modifier
IO$_LOADCODE	P1 - starting address of microcode to be loaded P2 - load byte count P3 - starting microprogram address to receive microcode	(none)
IO$_STARTMPROC	(none)	(none)
IO$_INITIALIZE	P1 - address of Initialize Command Table P2 - initialize command buffer length	(none)
IO _SETCLOCK	P2 - mode of operation P3 - clock control and status P4 - real-time clock preset value (2's complement)	(none)
IO$_STARTDATA	P1 - Data Transfer Command Table address P2 - Data Transfer Command Table length P3 - normal completion AST address P4 - overrun completion AST address	IO$M_SETEVF

High Level Language Subroutines	Functions
LPA$ADSWP	Start A/D converter sweep
LPA$DASWP	Start D/A converter sweep
LPA$DISWP	Start digital input sweep
LPA$DOSWP	Start digital output sweep
LPA$LAMSKS	Specify LPA11-K controller and digital mask words
LPA$SETADC	Specify channel select parameters
LPA$SETIBF	Specify buffer parameters
LPA$STPSWP	Stop sweep
LPA$CLOCKA	Set Clock A rate
LPA$CLOCKB	Set Closk B rate
LPA$XRATE	Compute clock rate and present value
LPA$IBFSTS	Return buffer status
LPA$IGTBUF	Return next available buffer
LPA$INXTBF	Alter buffer order
LPA$IWTBUF	Return next buffer or wait
LPA$RLSBUF	Release buffer to LPA11-K
LPA$RMVBUF	Remove buffer from device queue
LPA$CVADF	Convert A/D input to floating point
LPA$FLT16	Convert unsigned integer to floating point
LPA$LOADMC	Load microcode and initialize LPA-11K

A.10 DR32 DRIVER

QIO Functions	Arguments	Modifier
IO$_LOADMCODE	P1 - Starting address of microcode to be loaded P2 - load byte count	
IO$_STARTDATA	P1 - starting address of Data Transfer Command Table P2 - length of the Data Transfer Command Table	IO$M_SETEVF

High Level Language Subroutines	Function
XF$SETUP	Defines command and buffer areas; initializes queues
XF$STARTDEV	Issues a QIO that starts the DR32
XF$FREESET	Releases command packets onto FREEQ
XF$PKTBLD	Builds command packets; releases them onto INPTQ

XF$GETPKT Removes a command packet from TERMQ

XF$CLEANUP Deassigns the device channel and deallocates the command area

A.11 DUP11 DRIVER

Functions	Arguments	Modifiers
IO$ READLBLK IO$ READVBLK IO$ WRITELBLK IO$ WRITEVBLK	P1 - buffer address P2 - byte count	IO$M SRRUNOUT IO$M PTPBSC IO$M LASTBLOCK[1]
IO$ SETMODE	P1 - line parameters block	IO$M STARTUP IO$M NODSRWAIT[2] IO$M SHUTDOWN
IO$ SENSEMODE	(none)	

1. Only for write functions with IO$M PTPBSC

2. Use only with IO$M STARTUP

INDEX

VAX/VMS
I/O User's Guide
AA-D028B-TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please cut along this line.

Please indicate the type of reader that you most nearly represent.

- □ Assembly language programmer
- □ Higher-level language programmer
- □ Occasional programmer (experienced)
- □ User with little programming experience
- □ Student programmer
- □ Other (please specify)____________________

Name____________________ Date____________

Organization____________________

Street____________________

City____________ State__________ Zip Code or Country__________

— — — Do Not Tear - Fold Here and Tape —

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

— — Do Not Tear - Fold Here —