VMS I/O User's Reference Manual: Part I

Order Number: AA–LA84B–TE

June 1990

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

Revision/Update Information:	This document supersedes the *VMS I/O User's Reference Manual: Part I*, Version 5.0.
Software Version:	VMS Version 5.4

digital equipment corporation
maynard, massachusetts

June 1990

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA	DEQNA	MicroVAX	VAX RMS
DDIF	Desktop–VMS	PrintServer 40	VAXserver
DEC	DIGITAL	Q-bus	VAXstation
DECdtm	GIGI	ReGIS	VMS
DECnet	HSC	ULTRIX	VT
DECUS	LiveLink	UNIBUS	XUI
DECwindows	LN03	VAX	
DECwriter	MASSBUS	VAXcluster	digital™

The following are third-party trademarks:

IBM is a registered trademark of the International Business Machines Corporation.

PostScript is a registered trademark of Adobe Systems Incorporated.

ZK4513

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by Digital. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use Digital-supported devices, such as the LN03 laser printer and PostScript printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

FIGURES

TABLES

Preface

Intended Audience

This manual is intended for system programmers who want to take advantage of the time and space savings that result from direct use of I/O devices. Users of VMS who do not require such detailed knowledge of I/O drivers can use the device-independent services described in the *VMS Record Management Services Manual*.

Document Structure

This manual is organized into eleven chapters and three appendixes, as follows:

- Chapter 1 describes the Queue I/O (QIO) interface to file system ancillary control processes (ACPs).
- Chapters 2 through 11 describe the use of VMS file-structured and real-time I/O device drivers, the drivers for storage devices such as disks and magnetic tapes, and terminal devices supported by VMS:
  - Chapter 2 discusses the card reader driver.
  - Chapter 3 discusses disk drivers.
  - Chapter 4 discusses the LPA11-K driver.
  - Chapter 5 discusses the line printer drivers.
  - Chapter 6 discusses the magnetic tape drivers.
  - Chapter 7 discusses the mailbox driver.
  - Chapter 8 discusses the terminal driver.
  - Chapter 9 discusses the pseudoterminal driver.
  - Chapter 10 discusses the shadow-set virtual unit driver.
  - Chapter 11 discusses the VMS Generic Small Computer Systems Interface (SCSI) class driver.
- Appendix A summarizes the QIO function codes, arguments, and function modifiers used by the drivers listed above.
- Appendix B lists the DEC Multinational Character Set and the ANSI and DIGITAL-private escape sequences for terminals.
- Appendix C describes the VAX calling standards for the control connection routines.

Associated Documents

The following documents provide additional information:

- *VMS System Services Reference Manual*
- *VMS Software Information Management Handbook*
- *VMS Software VMS System Software Handbook*
- *Guide to VMS Programming Resources*
- *VMS Record Management Services Manual*
- *LPA11-K Laboratory Peripheral Accelerator User's Guide*
- *VMS Networking Manual*
- *VMS System Messages and Recovery Procedures Reference Manual*
- *VMS Device Support Manual*

Conventions

The following conventions are used in this manual:

Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 x	A sequence such as PF1 x indicates that you must first press and release the key labeled PF1, then press and release another key or a pointing device button.
Return	In examples, a key name is shown enclosed in a box to indicate that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	In examples, a horizontal ellipsis indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
()	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.

[]	In format descriptions, brackets indicate that whatever is enclosed within the brackets is optional; you can select none, one, or all of the choices. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
{}	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
red ink	Red ink indicates information that you must enter from the keyboard or a screen object that you must choose or click on. For online versions of the book, user input is shown in **bold**.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason. Boldface text is also used to show user input in online versions of the book.
italic text	Italic text represents information that can vary in system messages (for example, Internal error *number*).
UPPERCASE TEXT	Uppercase letters indicate that you must enter a command (for example, enter OPEN/READ), or they indicate the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.
-	Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows.
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1 ACP—QIO Interface

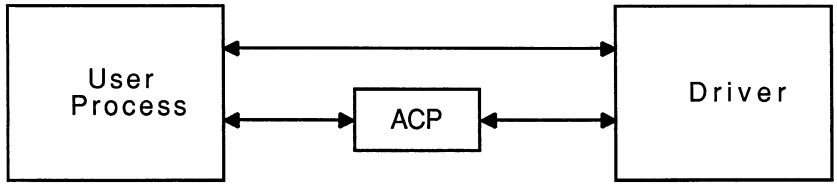
An ancillary control process (ACP) is a process that interfaces between the user process and the driver, and performs functions that supplement the driver's functions. Virtual I/O operations involving file-structured devices (disks and magnetic tapes) often require ACP intervention. In most cases, ACP intervention is requested by VMS 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 1–1.

Executing physical and logical I/O operations on a device being managed by a file ACP will interfere with the operation of the ACP and will result in unpredictable consequences, including system failure in certain cases.

In addition to the ACP, the VMS operating system also provides the XQP (extended QIO processor) facility to supplement the QIO driver's functions when performing virtual I/O operations on file-structured devices (ACP for Files–11 On-Disk Structure Level 1 and XQP for Files–11 On-Disk Structure Level 2). However, rather than being a separate process, the XQP executes as a kernel mode thread in the process of its caller.

This chapter describes the QIO interface to ACPs for disk and magnetic tape devices (file system ACPs). The sample program in Chapter 6 performs QIO operations to the magnetic tape ACP.

Figure 1–1 ACP–QIO Interface



ZK–0635–GE

This section also describes a number of structures and field names of the form xxx$name. A VAX MACRO program can define symbols of this form by invoking the $xxxDEF macro.

The following macros are available in SYS$LIBRARY:STARLET.MLB:

$IODEF
$FIBDEF
$ATRDEF
$SBKDEF

The following macros are available in SYS$LIBRARY:LIB.MLB:

$FATDEF
$DQFDEF
$FCHDEF

Programs written in BLISS-32 can use these symbols by referencing them and including the correct library, SYS$LIBRARY:STARLET.L32 (for the macros listed under SYS$LIBRARY:STARLET.MLB), and SYS$LIBRARY:LIB.L32 (for the macros listed under SYS$LIBRARY:LIB.MLB).

References to ANSI refer to the *American National Standard Magnetic Tape Labels and File Structures for Information Interchange, ANSI X3.27–1978.*

1.1 ACP Functions and Encoding

All VMS ACP functions can be expressed using seven function codes and four function modifiers. The function codes are as follows:

- IO$_CREATE—Creates a directory entry or file
- IO$_ACCESS—Searches a directory for a specified file and accesses the 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 file allocation
- IO$_DELETE—Deletes a directory entry and file header
- IO$_MOUNT—Informs the ACP when a volume is mounted; requires MOUNT privilege
- IO$_ACPCONTROL—Performs miscellaneous control functions

The function modifiers are:

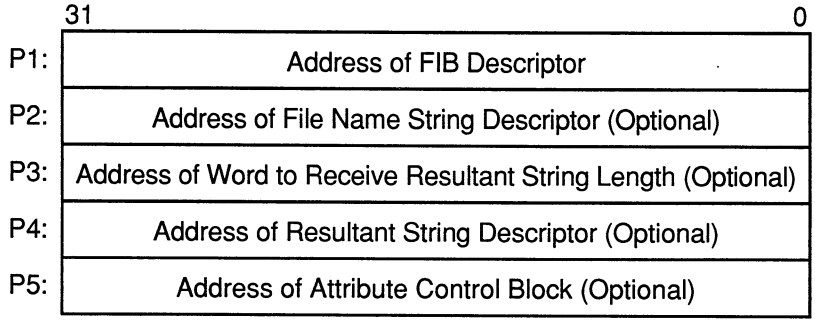
- IO$M_ACCESS—Opens a file on the user's channel
- IO$M_CREATE—Creates a file
- IO$M_DELETE—Deletes a file (or marks it for deletion)
- IO$M_DMOUNT—Dismounts a volume

In addition to the function codes and modifiers, VMS ACPs take five device- or function-dependent arguments, as shown in Figure 1–2. The first argument, P1, is the address of the file information block (FIB) descriptor. Section 1.2 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.

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 of these arguments are optional.

Figure 1–2 ACP Device- or Function-Dependent Arguments



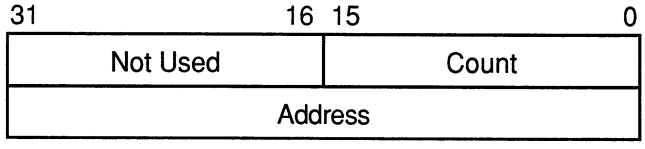
ZK–0636–GE

The fifth argument, P5, is an optional argument containing the address of the attribute control block. Section 1.3.5 describes the attribute control block in detail.

All areas of memory specified by the descriptors must be capable of being read or written to.

Figure 1–3 shows the format for the descriptors. The count field is the length in bytes of the item described.

Figure 1–3 ACP Device/Function Argument Descriptor Format



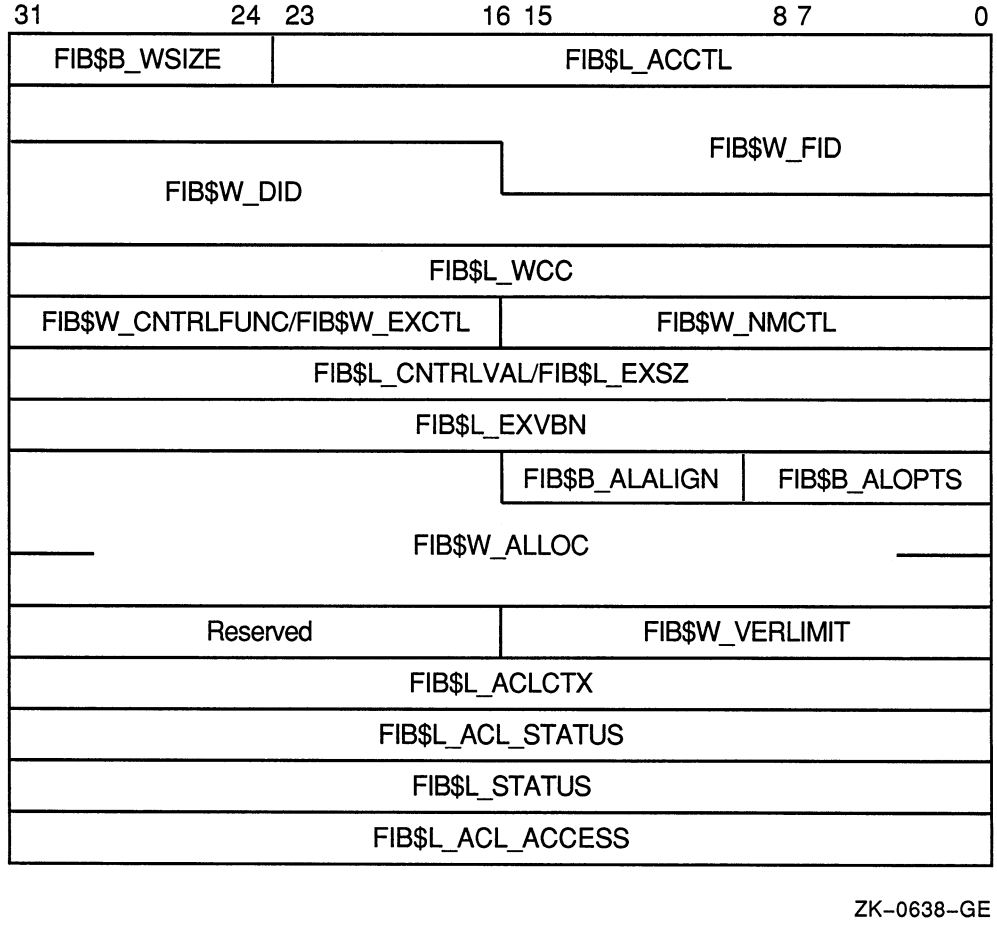
ZK–0637–GE

1.2 File Information Block (FIB)

The file information block (FIB) contains much of the information that is exchanged between the user process and the ACP. Figure 1–4 shows the format of the FIB. The FIB must be writable. Because the FIB is passed by a descriptor (see Figure 1–3), its length can vary. Thus, a short FIB can be used in ACP calls that do not need arguments near the end of the FIB. The ACP treats the omitted portion of the FIB as if it were 0. Figure 1–5 shows the format of a typical short FIB that would be used to open an

existing file. Table 1–1 gives a brief description of each of the FIB fields. More detailed descriptions are provided in Sections 1.3 and 1.6.

Figure 1–4 File Information Block Format



ZK–0638–GE

Figure 1–5 Typical Short File Information Block

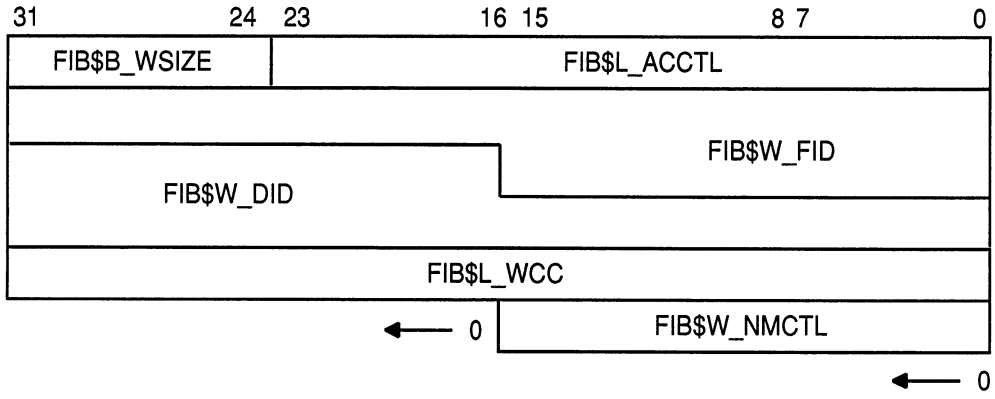


Table 1–1 Contents of the File Information Block

Field	Subfields	Meaning
FIB$L_ACCTL		Contains flag bits that control the access to the file. Sections 1.3.1.1, 1.3.2.1, 1.6.1.1, 1.6.4.1, and 1.6.5 describe the FIB$L_ACCTL field flag bits.
FIB$B_WSIZE		Controls the size of the file window used to map a disk file. If a window size of 255 is specified, the file is mapped completely through the use of segmented windows.
FIB$W_FID		Specifies the file identification. You supply 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. A 0 file identifier can be specified when an operation is performed on a file that is already open on a particular channel. The ACP returns the file identifier of the open file. The following subfields are defined:
	FIB$W_FID_NUM	File number.
	FIB$W_FID_SEQ	File sequence number.
	FIB$W_FID_RVN	Relative volume number (only for magnetic tape devices).
	FIB$B_FID_RVN	Relative volume number (only for disk devices).
	FIB$B_FID_NMX	File number extension (only for disk devices).
FIB$W_DID		Contains the file identifier of the directory file. The following subfields are defined:
	FIB$W_DID_NUM	File number.
	FIB$W_DID_SEQ	File sequence number.
	FIB$W_DID_RVN	Relative volume number (only for magnetic tape devices).

(continued on next page)

Table 1–1 (Cont.) Contents of the File Information Block

Field	Subfields	Meaning
	FIB$B_DID_RVN	Relative volume number (only for disk devices).
	FIB$B_DID_NMX	File number extension (only for disk devices).
FIB$L_WCC		Maintains position context when processing wildcard directory operations.
FIB$W_NMCTL		Contains flag bits that control the processing of a name string in a directory operation. Sections 1.3.1.1 and 1.6.1.1 describe the FIB$W_NMCTL field flag bits.
FIB$W_EXCTL		Contains flag bits that specify extend control for disk devices. Sections 1.3.3.1 and 1.3.4.1 describe the FIB$W_EXCTL field flag bits.
FIB$W_CNTRLFUNC		In an IO$_ACPCONTROL function, this field contains the code that specifies which ACP control function is to be performed (see Section 1.6.7). This field overlays FIB$W_EXCTL.
	FIB$C_USEREOT	User EOT mode. In an IO$_CREATE or IO$_ACCESS function, you can set this mode on a per-file basis. (See Sections 1.6.1 and 1.6.2.)
FIB$L_EXSZ		Specifies the number of blocks to be allocated in an extend operation on a disk file.
FIB$L_CNTRLVAL		Contains a control function value used in an IO$_ ACPCONTROL function (see Section 1.6.7). The interpretation of the value depends on the control function specified in FIB$W_CNTRLFUNC. This field overlays FIB$L_EXSZ.
FIB$L_EXVBN		Specifies the starting disk file virtual block number at which a file is to be truncated.
FIB$B_ALOPTS		Contains option bits that control the placement of allocated blocks. Section 1.3.3.1 describes the FIB$B_ALOPTS field flag bits.
FIB$B_ALALIGN		Contains the interpretation mode of the allocation (FIB$W_ALLOC) field.
FIB$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	Three-word related file ID for RFI placement.
	FIB$W_LOC_NUM	Related file number.
	FIB$W_LOC_SEQ	Related file sequence number.
	FIB$B_LOC_RVN	Related file RVN or placement RVN.
	FIB$B_LOC_NMX	Related file number extension.
	FIB$L_LOC_ADDR	Placement LBN, cylinder, or VBN.
FIB$W_VERLIMIT		Contains the version limit of the directory entry.
FIB$L_ACLCTX		Maintains position context when processing ACL attributes from the attribute (P5) list.

(continued on next page)

Table 1–1 (Cont.) Contents of the File Information Block

Field	Subfields	Meaning
FIB$L_ACL_STATUS		Status of the requested ACL attribute operation, if any. The ACL attributes are included in Table 1–7. If no ACL attributes are given, SS$_NORMAL is returned here.
FIB$L_STATUS		Access status. Applies to all major functions. The following bits are supported:
	FIB$V_ALT_REQ	Set to indicate whether the alternate access bit is required for the current operation. If not set, the alternate access bit is optional.
	FIB$V_ALT_GRANTED	If FIB$V_ALT_REQ = 0, the FIB bit returned from the file system is set if the alternate access check succeeded.
FIB$L_ALT_ACCESS		A 32-bit mask that represents an access mask to check against file protection; for example, opens a file for read access and checks whether it can be deleted. The mask has the same configuration as the standard protection mask.

1.3 ACP Subfunctions

The operations that the ACP performs can be organized into two categories: major ACP functions and subfunctions. Each ACP operation performs one major function. That function is specified by an I/O function code, such as IO$_ACCESS, IO$_CREATE, or IO$_MODIFY. While executing the major function, one or more subfunctions can be performed. A subfunction is an operation such as looking up, accessing, or extending a file. Most subfunctions can be executed by more than one of the major functions. Sections 1.3.1 through 1.3.5 describe the following subfunctions in detail:

- Directory Lookup
- Access
- Extend
- Truncate
- Read Attributes
- Write Attributes

Section 1.6, which contains the descriptions of the major functions, lists the subfunctions available to each major function.

1.3.1 Directory Lookup

The directory lookup subfunction is used to search for a file in a disk directory or on a magnetic tape. This subfunction can be invoked using the major functions IO$_ACCESS, IO$_MODIFY, IO$_DELETE, and IO$_ACPCONTROL. A directory lookup occurs if the directory file ID field in the FIB (FIB$W_DID) is a nonzero number.

1.3.1.1 Input Parameters

Table 1–2 lists the FIB fields that control the processing of a lookup subfunction.

Table 1–2 FIB Fields (Lookup Control)

Field	Field Values	Meaning
FIB$W_NMCTL		Name string control. The following name control bits are applicable to a lookup operation:
	FIB$M_WILD	Set if name string contains wildcards. Setting this bit causes wildcard context to be returned in FIB$L_WCC.
	FIB$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$M_FINDFID	Set to search a directory for the file identifier in FIB$W_FID.
FIB$W_FID		File identification. The file ID of the file found is returned in this field.
FIB$W_DID		Contains the file identifier of the directory file. This field must be a nonzero number.
FIB$L_WCC		Maintains position context when processing wildcard directory operations.
FIB$L_ACCTL		The following access control flag is applicable to a lookup subfunction:
	FIB$M_REWIND	Set to rewind magnetic tape before lookup. If not set, a magnetic tape is searched from its current position.

QIO arguments P2 through P6 are passed as values. The second argument, P2, specifies the address of the descriptor for the file name string to be searched for in the directory.

The file name string must have one of the following two formats:

name.type;version

name.type.version

The name and type can be any combination of alphanumeric characters, and the dollar sign ($), asterisk (*), and percent (%) characters. The version must consist of numeric characters optionally preceded by a minus sign (–) (only for disk devices) or a single asterisk. The total number of alphanumeric and percent characters in the name field and in the type field must not exceed 39. Any number of additional asterisks can be present.

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 are ignored and the contents are assumed to be an asterisk.

Note that the file name string cannot contain a directory string. The directory is specified by the FIB$W_DID field (see Table 1–1). Only VMS RMS can process directory strings.

Argument P3 is the address of a word to receive the resultant file name string length.

Argument P4 is the address of a descriptor for a buffer to receive the resultant file name string. The resultant string is not padded. The P3 and P4 arguments are optional.

1.3.1.2 Operation

The system searches either the directory file specified by FIB$W_DID or the magnetic tape for the file name specified in the P2 file name parameter. The actual file name found and its length are returned in the P3 and P4 length and result string buffers. The file ID of the file found is returned in FIB$W_FID and can be used in subsequent operations as the major function is processed.

Zero and negative version numbers have special significance in a disk lookup operation. Specifying 0 as a version number causes the latest version of the file to be found. Specifying –1 locates the second most recent version, –2 the third most recent, and so forth. Specifying a version of –0 locates the lowest numbered version of the file. For magnetic tape lookups, a version number of 0 locates the first occurrence of the file encountered; negative version numbers are not allowed.

Wildcard lookups are performed by specifying the appropriate wildcard characters in the name string and setting FIB$M_WILD. (The name control bits FIBM_ALLNAM, FIBM_ALLTYP, and FIB$M_ALLVER can also be used in searching for wildcard entries, but they are intended primarily for compatibility mode use.) On the first lookup, FIB$L_WCC should contain zero entries. On each lookup, the ACP returns a nonzero value in
FIB$L_WCC, which must be passed back on the next lookup call. In addition, you must pass the resultant name string returned by the previous lookup using the P4 result string buffer, and its length in the P3 result length word. This string is used together with FIB$L_WCC to continue the wildcard search at the correct position in the directory.

Perform a lookup by file ID by setting the name control bit FIB$M_ FINDFID. When this bit is set, the system searches the directory for an entry containing the file ID specified in FIB$W_FID, and the name of the entry found is returned in the P3 and P4 result parameters. Note that if a directory contains multiple entries with the same file ID, only the first entry can be located with this technique.

Lookups by file ID should be done only when the file name is not available, because lookups by this method are often significantly slower than lookups by file name.

1.3.1.3 Directory Entry Protection

A directory entry is protected with the same protection code as the file itself. For example, if a file is protected against delete access, then the file name has the same protection. Consequently, a nonprivileged user (that is, a user who is not the volume owner or a user who does not have SYSPRV) cannot rename a file because renaming a file is essentially the same as deleting the file name. This protection is applied regardless of the protection on the directory file.

Nonprivileged users can neither write directly into a .DIR;1 directory file nor turn off the directory bit in a directory file header.

1.3.2 Access

The access subfunction is used to open a file so that virtual read or write operations can be performed. This subfunction can be invoked using the major functions IO$_CREATE and IO$_ACCESS (see Sections 1.6.1 and 1.6.2). An access subfunction is performed if the IO$M_ACCESS modifier is specified in the I/O function code.

1.3.2.1 Input Parameters

Table 1–3 lists the FIB fields that control the processing of an access subfunction.

Table 1–3 FIB Fields (Access Control)

Field	Field Values	Meaning
FIB$L_ACCTL		Specifies field values that control access to the file. The following access control bits are applicable to the access subfunction:
	FIB$M_WRITE	Set for write access; clear for read-only access.
	FIB$M_NOREAD	Set to deny read access to others. (You must have write privilege to the file to use this option.)
	FIB$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). Used only for disk devices.
		Used to flag a file as inconsistent if the program currently modifying the file terminates abnormally. If the program deaccesses the file without performing a write attributes operation, the file is marked as locked and cannot be accessed until it is unlocked.
	FIB$M_UPDATE	Set to position at start of a magnetic tape file when opening file for write; clear to position at end-of-file.
	FIB$M_READCK	Set to enable read checking of the file. Virtual reads to the file are performed using a data check operation.
	FIB$M_WRITECK	Set to enable write checking of the file. Virtual writes to the file are performed using a data check operation.
	FIB$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 SUPERVISOR, EXEC, or KERNEL mode.

(continued on next page)

Table 1–3 (Cont.) FIB Fields (Access Control)

Field	Field Values	Meaning
	FIB$M_NOLOCK	Set to override exclusive access to the file, allowing you to access the file when another user has the file open with FIB$M_NOREAD specified. You must have SYSPRV privilege or ownership of the volume to use this option. FIB$M_NOREAD and FIB$M_NOWRITE must be clear for this option to work.
	FIB$M_ NORECORD	Set to inhibit recording of the file's expiration date. If not set, the file's expiration date can be modified, depending on the file retention parameters of the volume.
FIB$B_WSIZE		Controls the size of the file window used to map a disk file. The ACP uses the volume default if FIB$B_WSIZE is 0. A value of 1 to 127 indicates the number of retrieval pointers to be allocated to the window. A value of –1 indicates that the window should be as large as necessary to map the entire file. Note that the window is charged to the user's BYTELIM quota.
FIB$W_FID		Specifies the file identification of the file to be accessed.

1.3.2.2 Operation

The file is opened according to the access control specified (see Table 1–3).

1.3.3 Extend

The extend subfunction is used to allocate space to a disk file. This subfunction can be invoked using the major I/O functions IO$_CREATE and IO$_MODIFY (see Sections 1.6.1 and 1.6.4). The extend subfunction is performed if the bit FIB$M_EXTEND is set in the extend control word FIB$W_EXCTL.

1.3.3.1 Input Parameters

Table 1–4 lists the FIB fields that control the processing of an extend subfunction.

Table 1–4 FIB Fields (Extend Control)

Field	Field Values	Meaning
FIB$W_EXCTL		Extend control flags. The following flags are applicable to the extend subfunction:
	FIB$M_EXTEND	Set to enable extension.
	FIB$M_NOHDREXT	Set to inhibit generation of extension file headers.
	FIB$M_ALCON	Allocates contiguous space. The extend operation fails if the necessary contiguous space is not available.
	FIB$M_ALCONB	Allocates the maximum amount of contiguous space.

(continued on next page)

Table 1–4 (Cont.) FIB Fields (Extend Control)

Field	Field Values	Meaning
		If both FIB$M_ALCON and FIB$M_ALCONB are set, a single contiguous area, whose size is the largest available but not greater than the size requested, is allocated.
	FIB$M_FILCON	Marks the file contiguous. This bit can only be set if the file does not have space already allocated to it.
	FIB$M_ALDEF	Allocates the extend size (FIB$L_EXSZ) or the system default, whichever is greater.
FIB$L_EXSZ		Specifies the number of blocks to allocate to the file.
		The number of blocks actually allocated for this operation is returned in this longword. More blocks than requested can be allocated to meet cluster boundaries.
FIB$L_EXVBN		Returns the starting virtual block number of the blocks allocated. FIB$L_EXVBN must initially contain 0 blocks.
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. If this bit is set and the specified blocks are not available, the extend operation fails.
	FIB$M_ONCYL	Set to locate allocated space within a cylinder. This option functions correctly only when FIB$M_ALCON or FIB$M_ALCONB is specified.
FIB$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 byte relative volume number (RVN) in FIB$B_LOC_RVN and a cylinder number in FIB$L_LOC_ADDR.
	FIB$C_LBN	Location is specified as a byte RVN in FIB$B_LOC_RVN, followed by a longword logical block number (LBN) in FIB$L_LOC_ADDR.
	FIB$C_VBN	Location is specified as a longword virtual block number (VBN) of the file being extended in FIB$L_LOC_ADDR. A 0 VBN or one that fails to map indicates the end of the file.
	FIB$C_RFI	Location is specified as a three-word file ID in FIB$W_LOC_FID, followed by a longword VBN of that file in FIB$L_LOC_ADDR. A 0 file ID indicates the file being extended. A 0 VBN or one that fails to map indicates the end of that file.
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$W_LOC_FID	Three-word related file ID for RFI placement.
	FIB$W_LOC_NUM	Related file number.

(continued on next page)

Table 1–4 (Cont.) FIB Fields (Extend Control)

Field	Field Values	Meaning
	FIB$W_LOC_SEQ	Related file sequence number.
	FIB$B_LOC_RVN	Related file RVN or placement RVN.
	FIB$B_LOC_NMX	Related file number extension.
	FIB$L_LOC_ADDR	Placement LBN, cylinder, or VBN.

1.3.3.2 Operation

The specified number of blocks are allocated and appended to the file. The virtual block number assigned to the first block allocated is returned in FIB$L_EXVBN. The actual number of blocks allocated is returned in FIB$L_EXSZ.

The actual number of blocks allocated is also returned in the second longword of the user's I/O status block. If a contiguous allocation (FIB$M_ALCON) fails, the size of the largest contiguous space available on the disk is returned in the second longword of the user's I/O status block.

1.3.4 Truncate

The truncate subfunction is used to remove space from a disk file. This subfunction can be invoked by the major I/O functions IO$_DEACCESS and IO$_MODIFY (see Sections 1.6.3 and 1.6.4). The truncate subfunction is performed if the bit FIB$M_TRUNCATE is set in the extend control word FIB$W_EXCTL.

1.3.4.1 Input Parameters

Table 1–5 lists the FIB fields that control the processing of a truncate subfunction.

Table 1–5 FIB Fields (Truncate Control)

Field	Field Values	Meaning
FIB$W_EXCTL		Extend control flags. The following flags are applicable to the truncate subfunction:
	FIB$M_TRUNC	Must be set to enable truncation.
	FIB$M_MARKBAD	Set to append the truncated blocks to the bad block file, instead of returning them to the free storage pool. Only one cluster can be deallocated. This is most easily accomplished by specifying the last VBN of the file in FIB$L_EXVBN. SYSPRV privilege or ownership of the volume is required to deallocate blocks to the bad block file.
FIB$L_EXSZ		Returns the actual number of blocks deallocated. FIB$L_EXSZ must initially contain a value of 0.

(continued on next page)

Table 1–5 (Cont.) FIB Fields (Truncate Control)

Field	Field Values	Meaning
FIB$L_EXVBN		Specifies the first virtual block number to be removed from the file. The actual starting virtual block number of the truncate operation is returned in this field.

1.3.4.2 Operation

Blocks are deallocated from the file, starting with the virtual block specified in FIB$L_EXVBN and continuing through the end of the file. The actual number of blocks deallocated is returned in FIB$L_EXSZ. The virtual block number of the first block actually deallocated is returned in FIB$L_EXVBN. Because of cluster round-up, this value might be greater than the value specified. If FIB$M_MARKBAD is specified, the truncation VBN is rounded down instead of up, and the value returned in FIB$L_EXVBN might be less than that specified.

The number of blocks by which FIB$L_EXVBN was rounded up is returned in the second longword of the I/O status block.

The truncate subfunction normally requires exclusive access to the file at run time. This means, for example, that a file cannot be truncated while multiple writers have access to it.

An exception occurs when a truncate subfunction is requested for a write-accessed file that allows other readers. Although the truncate subfunction returns success status in this instance, the actual file truncation (the return of the truncated blocks to free storage) is deferred until the last reader deaccesses the file. If a new writer accesses the file after the truncate subfunction is requested, but before the last deaccess, the deferred truncation is ignored.

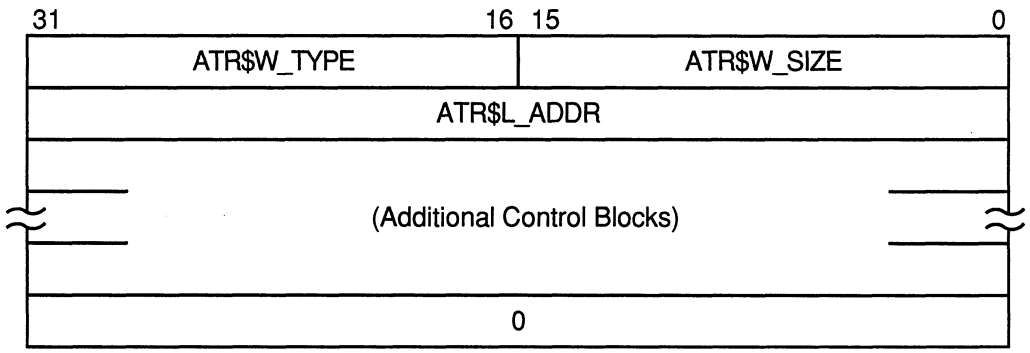
1.3.5 Read/Write Attributes

The read and write attributes subfunctions are used for operations such as reading and writing file protection and creating and revising dates. A read or write attributes operation is invoked by specifying an attribute list with the QIO parameter P5. A read attributes operation can be invoked by the major I/O function IO$_ACCESS (see Section 1.6.2); a write attributes operation can be invoked by the major I/O functions IO$_CREATE, IO$_DEACCESS, and IO$_MODIFY (see Sections 1.6.1, 1.6.3, and 1.6.4).

1.3.5.1 Input Parameters

The read or write attributes subfunction is controlled by the attribute list specified by P5. The list consists of a variable number of two longword control blocks, terminated by a 0 longword, as shown in Figure 1–6. The maximum number of attribute control blocks in one list is 30. Table 1–6 describes the attribute control block fields.

Figure 1–6 Attribute Control Block Format



ZK–0640–GE

Table 1–6 Attribute Control Block Fields

Field	Meaning
ATR$W_SIZE	Specifies the number of bytes of the attribute to be transferred. Legal values are from 0 to the maximum size of the particular attribute (see Table 1–7).
ATR$W_TYPE	Identifies the individual attribute to be read or written.
ATR$L_ADDR	Contains the buffer address of the memory space to or from which the attribute is to be transferred. The attribute buffer must be writable.

Table 1–7 lists the valid attributes for ACP-QIO functions. The maximum size (in bytes) is determined by the required attribute configuration. For example, the Radix–50 file name (ATR$S_FILNAM) uses only 6 bytes, but it is always accompanied by the file type and file version, so a total of 10 bytes is required. Each attribute has two names: one for the code (for example, ATR$C_UCHAR) and one for the size (for example, ATR$S_UCHAR).

Table 1–7 ACP—QIO Attributes

Attribute Name[1]	Maximum Size (bytes)	Meaning
ATR$C_UCHAR[2] [4]	4	4-byte file characteristics. (The file characteristics bits are listed following this table.)
ATR$C_RECATTR[3]	32	Record attribute area. Section 1.4 describes the record attribute area in detail.
ATR$C_FILNAM	10	6-byte Radix–50 file name plus ATR$C_FILTYP and ATR$C_FILVER.
ATR$C_FILTYP	4	2-byte Radix–50 file type plus ATR$C_FILVER.
ATR$C_FILVER	2	2-byte binary version number.
ATR$C_EXPDAT[2]	7	Expiration date in ASCII. Format: DDMMMYY.
ATR$C_STATBLK[5]	32	Statistics block. Section 1.5 describes the statistics block in detail.
ATR$C_HEADER[5]	512	Complete file header.
ATR$C_BLOCKSIZE	2	Magnetic tape block size.
ATR$C_USERLABEL[6]	80	User file label.
ATR$C_ASCDATES[2] [4]	35	Revision count (2 binary bytes), revision date, creation date, and expiration date, in ASCII. Format: DDMMMYY (revision date), HHMMSS (time), DDMMMYY (creation date), HHMMSS (time), DDMMMYY (expiration date). (The format contains no embedded spaces or commas.)
ATR$C_ALCONTROL	14	Compatibility mode allocation data.
ATR$C_ENDLBLAST	4	End of magnetic tape label processing; provides AST control block.

[1] Attributes with an ATR$C_ prefix have two names: one with the ATR$C_ prefix for the code and one with an ATR$S_ prefix for the size, which is not included in the list.

[2] Protected (can be written to only by system or owner).

[3] Locked (cannot be written to while the file is locked against writers).

[4] Not supported on write operations to MTAACP; defaults are returned on read operations.

[5] Read only.

[6] Not supported for disk devices.

(continued on next page)

Table 1–7 (Cont.) ACP—QIO Attributes

Attribute Name[1]	Maximum Size (bytes)	Meaning
ATR$C_ASCNAME	20	Disk: file name, type, and version, in ASCII, including punctuation. Format: name.type;version. Magnetic tape: contains 17-character file identifier (ANSI a); no version number. Overrides all other file name and file type specifications if supplied on input operations. If specified on an access operation and you want only a value to be returned, specify (in ATR$W_SIZE) a buffer of greater than 17 bytes.
ATR$C_CREDATE[2]	8	64-bit creation date and time.
ATR$C_REVDATE[2] [3]	8	64-bit revision date and time.
ATR$C_EXPDATE[2]	8	64-bit expiration date and time.
ATR$C_BAKDATE[3] [10]	8	64-bit backup date and time.
ATR$C_UIC[2]	4	4-byte file owner UIC.
ATR$C_FPRO[2] [3]	2	File protection.
ATR$C_RPRO[10]	2	2-byte record protection.
ATR$C_ACLEVEL[2] [3] [10]	1	File access level.
ATR$C_SEMASK[10]	8	File security mask and limit.
ATR$C_UIC_RO[5]	4	4-byte file owner UIC.
ATR$C_DIRSEQ[10]	2	Directory update sequence count.
ATR$C_BACKLINK[10]	6	File back link pointer.
ATR$C_JOURNAL[10]	2	Journal control flags.
ATR$C_HDR1_ACC	1	ANSI magnetic tape header label accessibility character.
ATR$C_ADDACLENT[7] [10] [11]	255	Add one or more access control entries.
ATR$C_DELACLENT[7] [10] [11]	255	Remove an access control entry.
ATR$C_MODACLENT[7] [10] [11]	255	Modify an ACL entry.

[1] Attributes with an ATR$C_ prefix have two names: one with the ATR$C_ prefix for the code and one with an ATR$S_ prefix for the size, which is not included in the list.

[2] Protected (can be written to only by system or owner).

[3] Locked (cannot be written to while the file is locked against writers).

[5] Read only.

[7] Exclusive access required. This operation does not complete successfully if other readers or writers are allowed.

[10] Not supported for Files–11 On-Disk Structure Level 1 or magnetic tapes.

[11] The status from this attribute operation is returned in FIB$L_ACL_STATUS.

(continued on next page)

Table 1–7 (Cont.) ACP—QIO Attributes

Attribute Name[1]	Maximum Size (bytes)	Meaning
ATR$C_FNDACLENT[10 11]	255	Locate an ACL entry.
ATR$C_FNDACETYP[10 11]	255	Find a specific type of ACE.
ATR$C_DELETEACL[7 10 11]	255	Delete the entire ACL, retaining any unprotected entries.
ATR$C_READACL[10 11]	512	Read the entire ACL or as much as will fit in the supplied buffer. Only complete ACEs are transferred. Thus, the supplied buffer can not be completely filled.
ATR$C_ACLLENGTH[10 11]	4	Return the length of the ACL.
ATR$C_READACE[10 11]	255	Read a single ACE.
ATR$C_RESERVED[9 10]	380	Modify reserve area.
ATR$C_HIGHWATER[10]	4	High-water mark (user read-only).
ATR$C_PRIVS_USED[8 10]	4	Privileges used to gain access.
ATR$C_MATCHING_ACE[8 10]	255	ACE used to gain access (if any).
ATR$C_ACCESS_MODE	1	Access mode for following attribute descriptors.
ATR$C_FILE_SPEC[10]	512	Convert FID to file specification.
ATR$C_BUFFER_OFFSET[4]	2	Offset length for ANSI magnetic tape header label buffer.
ATR$C_DELETE_ALL[7 10 11]	255	Delete the entire ACL.
ATR$C_GRANT_ACE[10 11]	255	Return an ACE which grants or denies access.
ATR$C_NEXT_ACE[10 11]	4	Step on to point to the next ACE in the ACL.

[1] Attributes with an ATR$C_ prefix have two names: one with the ATR$C_ prefix for the code and one with an ATR$S_ prefix for the size, which is not included in the list.

[4] Not supported on write operations to MTAACP; defaults are returned on read operations.

[7] Exclusive access required. This operation does not complete successfully if other readers or writers are allowed.

[8] This attribute can only be retrieved on the initial file access or create operation.

[9] The actual length available can decrease if the file is extended in a noncontiguous manner or if an ACL is applied to the file.

[10] Not supported for Files–11 On-Disk Structure Level 1 or magnetic tapes.

[11] The status from this attribute operation is returned in FIB$L_ACL_STATUS.

Table 1–8 lists the bits contained in the file characteristics longword, which is read with the ATR$C_UCHAR attribute.

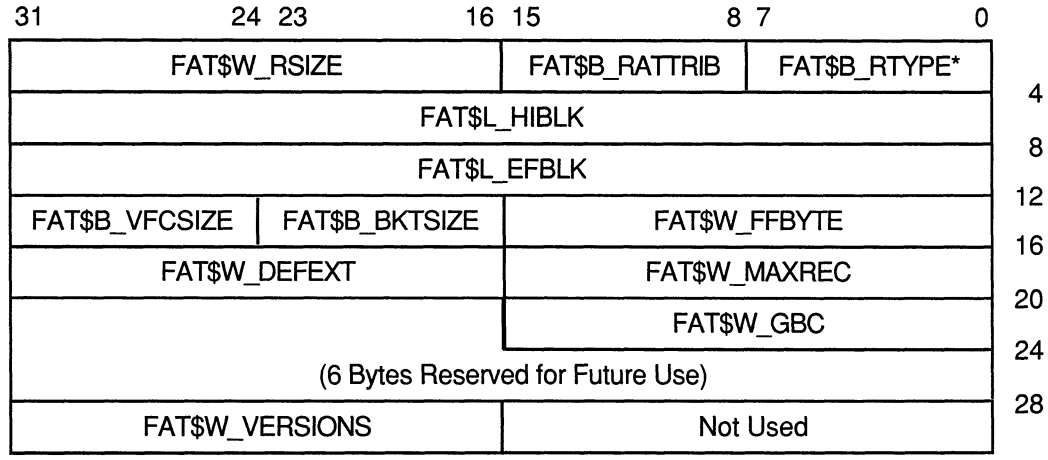
Table 1–8 File Characteristics Bits

FCHNOBACKUP	File is not to be backed up.
FCH$M_READCHECK	Verify all read operations.
FCH$M_WRITCHECK	Verify all write operations.
FCH$M_CONTIGB	Keep file as contiguous as possible.
FCH$M_LOCKED	File is deaccess-locked.
FCH$M_CONTIG	File is contiguous.
FCH$M_BADACL	File's ACL is corrupt.
FCH$M_SPOOL	File is an intermediate spool file.
FCH$M_DIRECTORY	File is a directory.
FCH$M_BADBLOCK	File contains bad blocks.
FCH$M_MARKDEL	File is marked for deletion.
FCH$M_ERASE	Erase file contents before deletion.

1.4 ACP QIO Record Attributes Area

Figure 1–7 shows the format of the record attributes area.

Figure 1–7 ACP—QIO Record Attributes Area



*FAT$V_RTYPE Bits 0–3; FAT$V_FILEORG Bits 4–7

ZK-0641-GE

1.4 ACP QIO Record Attributes Area

Table 1–9 lists the record attributes values and their meanings.

Table 1–9 ACP Record Attributes Values

Field Value	Meaning	
FAT$B_RTYPE	Record type. Contains FAT$V_RTYPE and FAT$V_FILEORG.	
FAT$V_RTYPE	Record type. The following bit values are defined:	
	FAT$C_FIXED	Fixed-length record
	FAT$C_VARIABLE	Variable-length record
	FAT$C_VFC	Variable-length record with fixed control
	FAT$C_UNDEFINED	Undefined record format (stream binary)
	FAT$C_STREAM	RMS stream format
	FAT$C_STREAMLF	Stream terminated by LF
	FAT$C_STREAMCR	Stream terminated by CR
FAT$V_FILEORG	File organization. The following bit values are defined:	
	FAT$C_DIRECT	Direct file organization[1]
	FAT$C_INDEXED	Indexed file organization
	FAT$C_RELATIVE	Relative file organization
	FAT$C_SEQUENTIAL	Sequential file organization
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$L_HIBLK[2]	Highest allocated VBN. The ACP maintains this field when the file is extended or truncated. Attempts to modify this field in a write attributes operation are ignored.	
	FAT$W_HIBLKH	High-order 16 bits
	FAT$W_HIBLKL	Low-order 16 bits
FAT$L_EFBLK[2 3]	End-of-file VBN	
	FAT$W_EFBLKH	High-order 16 bits
	FAT$W_EFBLKL	Low-order 16 bits
FAT$W_FFBYTE[3]	First free byte in FAT$L_EFBLK.	
FAT$B_BKTSIZE	Bucket size in blocks.	

[1]Defined but not implemented.

[2]Inverted format field. The high- and low-order 16 bits are transposed for compatibility with PDP–11 software.

[3]When the end-of-file position corresponds to a block boundary, by convention FAT$L_EFBLK contains the end-of-file VBN plus 1, and FAT$W_FFBYTE contains 0.

(continued on next page)

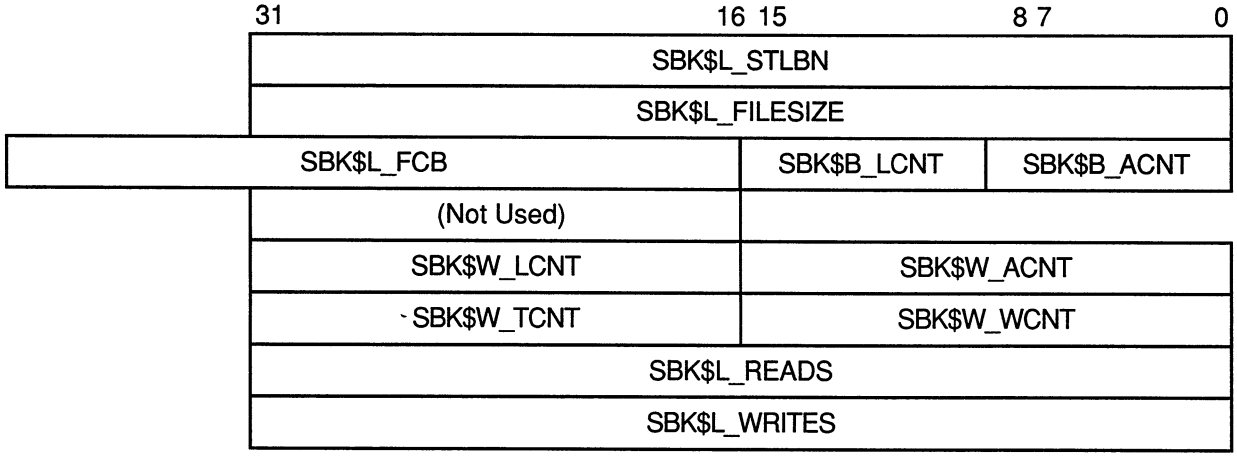
Table 1–9 (Cont.) ACP Record Attributes Values

Field Value	Meaning
FAT$B_VFCSIZE	Size in bytes of fixed-length control for VFC records.
FAT$W_MAXREC	Maximum record size in bytes.
FAT$W_DEFEXT	Default extend quantity.
FAT$W_GBC	Global buffer count.
FAT$W_VERSIONS	Default version limit; valid only if the file is a directory.

1.5 ACP–QIO Attributes Statistics Block

Figure 1–8 shows the format of the attributes statistics block. Table 1–10 lists the contents of this block.

Figure 1–8 ACP–QIO Attributes Statistics Block



ZK–0642–GE

Table 1–10 Contents of the Statistics Block

Field	Field Values	Meaning
SBK$L_STLBN		Contains the starting LBN of the file if the file is contiguous. If the file is not contiguous, this field contains a value of 0. The LBN appears as an inverted longword (the high- and low-order 16 bits are transposed for PDP–11 compatibility). The following subfields are defined:

(continued on next page)

Table 1–10 (Cont.) Contents of the Statistics Block

Field	Field Values	Meaning
	SBK$W_STLBNH	Starting LBN (high-order 16 bits).
	SBK$W_STLBNL	Starting LBN (low-order 16 bits).
SBK$L_FILESIZE		Contains the size of the file in blocks. The file size appears as an inverted longword (the high- and low-order 16 bits are transposed for PDP–11 compatibility). The following subfields are defined:
	SBK$W_FILESIZH	File size (high-order 16 bits).
	SBK$W_FILESIZL	File size (low-order 16 bits).
SBK$B_ACNT[1]		Access count (low byte). Field is for PDP–11 compatibility.
SBK$B_LCNT[1]		Lock count (low byte). Field is for PDP–11 compatibility.
SBK$L_FCB		System pool address of the file's file control block.
SBK$W_ACNT[1]		Access count (number of channels with file open currently).
SBK$W_LCNT[1]		Lock count (the number of access operations that have locked the file against writers).
SBK$W_WCNT[1]		Writer count (the number of channels that currently have the file open for write).
SBK$W_TCNT[1]		Truncate lock count (the number of access operations that have locked the file against truncation).
SBK$L_READS		Number of read operations executed for file on this channel.
SBK$L_WRITES		Number of write operations executed for file on this channel.

[1]Accesses from processes on the local node in a cluster are counted.

1.6 Major Functions

The following sections describe the operation of the major ACP functions. Each section describes the required and optional parameters for a particular function, as well as the sequence in which the function is performed. For clarity, when a major function invokes a subfunction, the input parameters used by the subfunction are omitted.

1.6.1 Create File

Create file is a virtual I/O function that creates a directory entry or a file on a disk device, or a file on a magnetic tape device.

The following is the function code:

- IO$_CREATE

The following are the function modifiers:

- IO$M_CREATE—Creates a file.
- IO$M_ACCESS—Opens the file on your channel.
- IO$M_DELETE—Marks the file for deletion (applicable only to disk devices).

1.6.1.1 Input Parameters

The following are the device- or function-dependent arguments for IO$_CREATE:

- 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).

Table 1–11 lists fields in the FIB that are applicable to the IO$_CREATE operation.

Table 1–11 IO$_CREATE and the File Information Block

Field	Field Values	Meaning
FIB$L_ACCTL		Specifies field values that control access to the file. The following bits are applicable to the IO$_CREATE function:
	FIB$M_REWIND	Set to rewind magnetic tape before creating the file. Any data currently on the tape is overwritten.
	FIB$M_CURPOS	Set to create magnetic tape file at the current tape position. (Note: a magnetic tape file is 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$M_CURPOS creates a file at the next file position. Any data currently on the tape past the current file position is overwritten.
	FIB$M_WRITETHRU	Specifies that the file header is to be written back to the disk. If not specified and the file is opened, writing of the file header can be deferred to some later time.
FIB$W_CNTRLFUNC		Specifies the following value, which allows you to control actions subsequent to EOT detection on a magnetic tape file.

(continued on next page)

Table 1–11 (Cont.) IO$_CREATE and the File Information Block

Field	Field Values	Meaning
	FIB$C_USEREOT	Set on a per-file basis to specify user EOT mode. If this bit is set, user EOT handling is enabled. When writing, if EOT has been detected (considered a "serious exception") and user EOT handling is enabled, then the magnetic tape system returns the alternate success code SS$_ENDOFTAPE. When reading, if EOV is reached, then the alternate success code SS$_ENDOFVOLUME is returned. In either case, all subsequent I/O requests for the volume are completed with a failure status return of SS$_SERIOUSEXCP. The driver does not execute any I/O functions until the serious exception has been explicitly cleared by issuing an IO$_ACPCONTROL function (see Section 1.6.7). If the file is deaccessed or closed, the user EOT mode is cleared after further processing of the magnetic tape.
FIB$W_FID		Contains the file ID of the file created or entered.
FIB$W_DID		Contains the file identifier of the directory file.
FIB$W_NMCTL		Controls the processing of the file name in a directory operation. The following bits are applicable to the IO$_CREATE function:
	FIB$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_LOWVER	Set on return if a lower numbered version of the file exists. Only for disk devices.
	FIB$M_HIGHVER	Set on return if a higher numbered version of the file exists. Only for disk devices.
FIB$W_VERLIMIT		Specifies the version limit for the directory entry created. Used only for disk devices and only when the first version of a new file is created. If 0, the directory default is used. If a directory operation was performed, FIB$W_VERLIMIT always contains the actual version limit of the file.
FIB$L_ACL_STATUS		Status of the requested ACL attribute operation, if any. The ACL attributes are included in Table 1–7. If no ACL attributes are given, SS$_NORMAL is returned here.

1.6.1.2 Disk ACP Operation

If the modifier IO$M_CREATE is specified, a file is created. The file ID of the file created is returned in FIB$W_FID. If the modifier IO$M_DELETE is specified, the file is marked for deletion.

If a nonzero directory ID is specified in FIB$W_DID, a directory entry is created. The file name specified by parameter P2 is entered in the directory, together with the file ID in FIB$W_FID. (Section 1.3.1.1 describes the format for the file name string.) Wildcards are not permitted. Negative version numbers are treated as equivalent to a 0 version number. If a result string buffer and length are specified by P3 and P4, the actual file name entered, and its length, are returned.

The version number of the file receives the following treatment:

- If the version number in the specified file name is 0 or negative, the directory entry created gets a version number one greater than the highest previously existing version of that file (or version 1 if the file did not previously exist).
- If the version number in the specified file name is a nonzero number and FIB$M_NEWVER is set, the directory entry created gets a version number one greater than the highest previously existing version of that file, or the specified version number, whichever is greater.
- If the version number in the specified file name is a nonzero number and the directory already contains a file of the same name, type, and version, the previously existing file is set aside for deletion if FIB$M_SUPERSEDE is specified. If FIB$M_SUPERSEDE is not specified, the create operation fails with an SS$_DUPFILNAM status.
- If, after creating the new directory entry, the number of versions of the file exceeds the version limit, the lowest numbered version is set aside for deletion.
- If the file did not previously exist, the new directory entry is given a version limit as follows: the version limit is taken from FIB$W_VERLIMIT if it is a nonzero number; if it is 0, the version limit is taken from the default version limit of the directory file; if the default version limit of the directory file is 0, the version limit is set to 32,767 (the highest possible number).

The file name string entered in the directory is returned using the P3 and P4 result string parameters, if present. The file name string is also written into the header. If no directory operation was requested (FIB$W_DID is 0), the file name string specified by P2, if any, is written into the file header.

If an attribute list is specified by P5, a write attributes subfunction is performed (see Section 1.3.5).

If the modifier IO$M_ACCESS is specified, the file is opened (see Section 1.3.2).

If the extend enable bit FIB$M_EXTEND is specified in the FIB, an extend subfunction is performed (see Section 1.3.3).

Finally, if a file was set aside for deletion (IO$M_DELETE is specified), that file is deleted. If the file is deleted because the FIB$M_SUPERSEDE bit was set, the alternate success status SS$_SUPERSEDE is returned in the I/O status block. If the file is deleted because the version limit was exceeded, the alternate success status SS$_FILEPURGED is returned.

If an error occurs in the operation of an IO$_CREATE function, all actions performed to that point are reversed (the file is neither created nor changed), and the error status is returned to the user in the I/O status block.

1.6.1.3 Directory Entry Creation

Creating a new version of a file eliminates default access to the previously highest version of the file. For example, creating RESUME.TXT;4 masks RESUME.TXT;3 so that the DCL command TYPE RESUME.TXT yields the contents of version 4, not version 3. To protect the contents of the earlier version of a file, the creator of a file must have write access to the previous version of a file of the same name.

1.6.1.4 Magnetic Tape ACP Operation

No operation is performed unless the IO$M_CREATE modifier is specified. The magnetic tape is positioned as specified by FIB$M_REWIND and FIB$M_CURPOS, and the file is created. The name specified by the P2 parameter is written into the file header label.

If P5 specifies an attribute list, a write attributes subfunction is performed (see Section 1.3.5).

If the modifier IO$M_ACCESS is specified, the file is opened (see Section 1.3.2).

1.6.2 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 following is the function code:

- IO$_ACCESS

The following are the function modifiers:

- IO$M_CREATE—Creates a file.
- IO$M_ACCESS—Opens the file on your channel.

1.6.2.1 Input Parameters

The following are the device- or function-dependent arguments for IO$_ACCESS:

- 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).

Table 1–12 lists FIB fields that are applicable to the IO$_ACCESS operation.

Table 1–12 IO$_ACCESS and the File Information Block

Field	Field Values	Meaning
FIB$W_CNTRLFUNC		Specifies the value that allows the user to control actions subsequent to EOT detection on a magnetic tape file.
	FIB$C_USEREOT	Set on a per-file basis to specify user EOT mode. If this bit is set, the magnetic tape driver notifies the magnetic tape system when EOT has been detected (considered a "serious exception") when a file is accessed. In turn, the magnetic tape system returns the alternate success code SS$_ENDOFTAPE or SS$_ENDOFVOLUME. All subsequent I/O requests are completed with a failure status return of SS$_SERIOUSEXP. The driver does not execute any I/O functions until the serious exception has been explicitly cleared by issuing an IO$_ACPCONTROL function (see Section 1.6.7). If the file is deaccessed or closed, the user EOT mode is cleared after further processing of the magnetic tape.
FIB$W_VERLIMIT		Receives the version limit for the file. Applicable only if FIB$W_DID is a nonzero number (if a directory lookup is done). Used only for disk devices.
FIB$L_ACL_STATUS		Status of the requested ACL attribute operation, if any. The ACL attributes are included in Table 1–7. If no ACL attributes are given, SS$_NORMAL is returned here.
FIB$L_STATUS		Alternate access status. The following bits are supported:
	FIB$V_ALT_REQ	Set to indicate whether the alternate access bit is required for the current operation. If not set, the alternate access bit is optional.
	FIB$V_ALT_ GRANTED	If FIB$V_ALT_REQ = 0 and the alternate access check succeeded, the FIB bit returned from the file system is set.
FIB$L_ALT_ACCESS		A 32-bit mask that represents an access mask to check against file protection; for example, to open a file for read and to check whether it can be deleted. The mask has the same configuration as the standard protection mask.

1.6.2.2 Operation

If a nonzero directory file ID is specified in FIB$W_DID, a lookup subfunction is performed (see Section 1.3.1.) The version limit of the file found is returned in FIB$W_VERLIMIT.

If the directory search fails with a 'file not found' condition and the IO$M_CREATE function modifier is specified, the function is reexecuted as a CREATE. In that case, the argument interpretations for IO$_CREATE, rather than those for IO$_ACCESS, apply.

If IO$M_ACCESS is specified, an access subfunction is performed to open the file (see Section 1.3.2).

If P5 specifies an attribute list, a read attributes subfunction is performed (see Section 1.3.5).

1.6.3 Deaccess File

Deaccess file is a virtual I/O function that deaccesses a file and, if specified, writes final attributes in the file header.

The following is the function code:

- IO$_DEACCESS

IO$_DEACCESS takes no function modifiers.

1.6.3.1 Input Parameters

The following are the device- or function-dependent arguments for IO$_DEACCESS:

- P1—The address of the file information block (FIB) descriptor.
- P5—The address of a list of attribute descriptors (optional).

The following FIB field is applicable to a IO$_DEACCESS function:

Field	Meaning
FIB$W_FID	File identification of the file being deaccessed. This field can contain a value of 0. If it does not, it must match the file identifier of the accessed file.
FIB$L_ACL_STATUS	Status of the requested ACL attribute operation, if any. The ACL attributes are included in Table 1–7. If no ACL attributes are given, SS$_NORMAL is returned here.

1.6.3.2 Operation

For disk files, if P5 specifies an attribute control list and the file was accessed for a write operation, a write attributes subfunction is performed (see Section 1.3.5). If the file was opened for write, no attributes were specified, and 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.

For disk files, if the truncate enable bit FIB$M_TRUNCATE is specified in the FIB, a truncate subfunction is performed (see Section 1.3.4).

Finally, the file is closed. Trailer labels are written for a magnetic tape file that was opened for write.

1.6.4 Modify File

Modify file is a virtual I/O function that modifies the file attributes or allocation of a disk file. The IO$_MODIFY function is not applicable to magnetic tape.

The following is the function function code:

- IO$_MODIFY

IO$_MODIFY takes no function modifiers.

1.6.4.1 Input Parameters

The following are the device- or function-dependent arguments for IO$_MODIFY:

- 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).

The following FIB fields are applicable to the IO$_MODIFY function:

Field	Field Values	Meaning
FIB$L_ACCTL		Specifies field values that control access to the file. The following bits are applicable to the IO$_MODIFY function:
	FIB$M_WRITETHRU	Specifies that the file header is to be written back to the disk. If not specified and the file is currently open, writing of the file header can be deferred to some later time.
FIB$W_VERLIMIT		If a nonzero number, specifies the version limit for the file.
FIB$L_ACL_STATUS		Status of the requested ACL attribute operation, if any. The ACL attributes are included in Table 1–7. If no ACL attributes are given, SS$_NORMAL is returned here.

1.6.4.2 Operation

If a nonzero directory ID is specified in FIB$W_DID, a lookup subfunction is executed (see Section 1.3.1). If a nonzero version limit is specified in FIB$W_VERLIMIT and the directory entry found is the latest version of that file, the version limit is set to the value specified.

If P5 specifies an attribute list, a write attributes subfunction is performed (see Section 1.3.5).

The file can be either extended or truncated. If FIB$M_EXTEND is specified in the FIB, an extend subfunction is performed (see Section 1.3.3). If FIB$M_TRUNCATE is specified in the FIB, a truncate subfunction is performed (see Section 1.3.4). Extend and truncate operations cannot be performed at the same time.

1.6.5 Delete File

Delete file is a virtual I/O function that removes a directory entry or file header from a disk volume.

The following is the function code:

- IO$_DELETE

The following is the function modifier:

- IO$M_DELETE—Deletes the file (or marks it for deletion).

The following are the device- or function-dependent arguments for IO$_DELETE:

- 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).

The following FIB fields are applicable to the IO$_DELETE function:

Field	Field Values	Meaning
FIB$L_ACCTL		Specifies field values that control access to the file. The following bit is applicable to the IO$_DELETE function:
	FIB$M_WRITETHRU	Specifies that the file header is to be written back to the disk. If not specified and the file is currently open, writing of the file header can be deferred to some later time.
FIB$W_FID		Specifies the file identification to be deleted.

1.6.5.1 Operation

If a nonzero directory ID is specified in FIB$W_DID, a lookup subfunction is performed (see Section 1.3.1). The file name located is removed from the directory.

If the function modifier IO$M_DELETE is specified, the file is marked for deletion. If the file is not currently open, it is deleted immediately. If the file is open, it is deleted when the last accessor closes it.

1.6.6 Mount

Mount is a virtual I/O function that informs the ACP when a disk or magnetic tape volume is mounted. MOUNT privilege is required. IO$_MOUNT takes no arguments or function modifiers. This function is a part of the volume mounting operation only, and it is not meant for general use. Most of the actual processing is performed by the MOUNT command or the Mount Volume ($MOUNT) system service.

1.6.7 ACP Control

ACP Control is a virtual I/O function that performs miscellaneous control functions, depending on the arguments specified.

The following is the function code:

- IO$_ACPCONTROL

The following is the function modifier:

- IO$M_DMOUNT—Dismounts a volume.

1.6.7.1 Input Parameters

The following are the device- or function-dependent arguments for IO$_ACPCONTROL:

- 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).

Table 1–13 lists FIB fields that control the processing of the IO$_ACPCONTROL function.

Table 1–13 IO$_ACPCONTROL and the File Information Block

Field	Field Values	Meaning
FIB$W_CNTRLFUNC		Specifies the control function to be performed. This field overlays FIB$W_EXCTL.
FIB$L_CNTRLVAL		Specifies additional function-dependent data. This field overlays FIB$L_EXSZ.
FIB$L_ACL_STATUS		Status of the requested ACL attribute operation, if any. The ACL attributes are included in Table 1–7. If no ACL attributes are given, SS$_NORMAL is returned here.
FIB$L_STATUS		Alternate access status. The following bits are supported:
	FIB$V_ALT_REQ	Set to indicate whether the alternate access bit is required for the current operation. If not set, the alternate access bit is optional.
	FIB$V_ALT_GRANTED	If FIB$V_ALT_REQ = 0 and the alternate access check succeeded, the FIB bit returned from the file system is set.
FIB$L_ALT_ACCESS		A 32-bit mask that represents an access mask to check against file protection; for example, to open a file for read and to check whether it can be deleted or not. The mask has the same configuration as the standard protection mask.

1.6.7.2 Magnetic Tape Control Functions

Table 1–14 lists FIB field applicable to magnetic tape operations.

Table 1–14 Magnetic Tape Operations and the File Information Block

Field	Field Values	Meaning
FIB$W_CNTRLFUNC		Several ACP control functions are used for magnetic tape positioning. These functions are specified by supplying a FIB with P1 containing the FIB descriptor address. Modifiers and parameters P2, P3, and P4 are not allowed. These functions clear serious exceptions in magnetic tape drivers. The following control functions can be specified to control magnetic tape positioning:
	FIB$C_REWINDFIL	Rewind to beginning-of-file.
	FIB$C_REWINDVOL	Rewind to beginning-of-volume set.
	FIB$C_POSEND	Position to end-of-volume set.
	FIB$C_NEXTVOL	Force next volume.
	FIB$C_SPACE	Space n blocks forward or backward. The FIB$L_CNTRLVAL field specifies the number of magnetic tape blocks to space forward if positive or to space backward if negative.
	FIB$C_CLSEREXCP	If set, clears the serious exception in the magnetic tape driver (see FIB$C_USEREOT in Section 1.6.1 and Section 1.6.2). If writing, this allows the user to write data blocks beyond the EOT marker, which can result in the magnetic tape not conforming to the ANSI standard for magnetic tapes (see ANSI Standard X3.27–1978). If reading, this allows the user to handle the move to the next volume or to stop reading the tape. The user should not attempt to read past EOV.

1.6.7.3 Miscellaneous Disk Control Functions

Several ACP control functions are available for disk volume control. The following function does not use parameters P2, P3, and P4:

IO$M_DMOUNT — Specifying the dismount modifier on the IO$_ACPCNTRL function executes a dismount QIO. No parameters in the FIB are used; the FIB can be omitted. This function does not perform a dismount by itself, but is used to synchronize the ACP with the DISMOUNT command and the Dismount Volume ($DISMOUNT) system service.

The FIB$W_CNTRLFUNC field of the FIB specifies the following miscellaneous control functions (with no modifier on the IO$_ ACPCONTROL function code). These functions use no other parameters.

FIB$C_REMAP — Remap a file. The file window for the file open on the user's channel is remapped so that it maps the entire file.

FIB$C_LOCK_VOL	Allocation lock the volume. Operations that change the file structure, such as file creation, deletion, extension, and deaccess, are not permitted. If such requests are queued to the file system for an allocation-locked volume, they are not processed until the FIB$C_UNLK_VOL function is issued to unlock the volume. To issue the FIB$C_LOCK_VOL function, you must have either a system UIC or SYSPRV privilege, or be the owner of the volume.
FIB$C_UNLK_VOL	Unlock the volume. Cancels FIB$C_LOCK_VOL. To issue this function, you must have either a system UIC or SYSPRV privilege, or be the owner of the volume.

1.6.7.4 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. This section describes the control functions that operate on the quota file.

Table 1–15 lists the enable and disable quota control functions.

Table 1–15 Disk Quota Functions (Enable/Disable)

Value	Meaning
FIB$C_ENA_QUOTA	Enable the disk quota file. If a nonzero directory file ID is specified in FIB$W_DID, a lookup subfunction is performed to locate the quota file (see Section 1.3.1). To issue this function, you must have either a system UIC or SYSPRV privilege, or be the owner of the volume. The quota file specified by FIB$W_FID, if present, is accessed by the ACP, and quota enforcement is turned on. By convention, the quota file is named [0,0]QUOTA.SYS;1. Therefore, FIB$W_DID should contain the value 4,4,0 and the name string specified with P2 should be "QUOTA.SYS;1".
FIB$C_DSA_QUOTA	Disable the disk quota file. The quota file is deaccessed and quota enforcement is turned off. To issue this function, you must have either a system UIC or SYSPRV privilege, or be the owner of the volume.

Table 1–16 lists the quota control functions that operate on individual entries in the quota file. Each operation transfers quota file data to and from the ACP using a quota data block. This block has the same format as a record in the quota file. Figure 1–9 shows the format of this block.

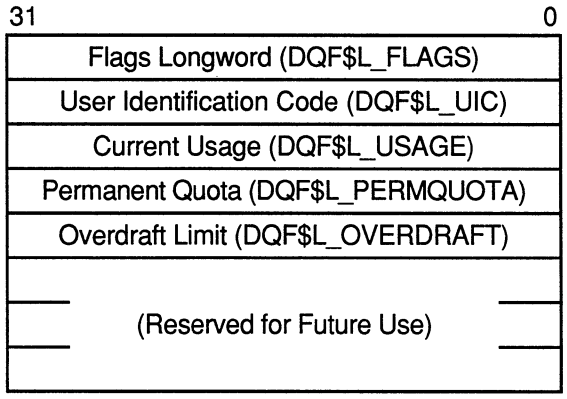
IO$_ACPCONTROL functions that transfer quota file data between the caller and the ACP use the following device- or function-dependent arguments:

- P2—The address of a descriptor for the quota data block being sent to the ACP.
- P3—The address of a word that returns the data length.
- P4—The address of a descriptor for a buffer to receive the quota data block returned from the ACP.

Table 1–16 Disk Quota Functions (Individual Entries)

Value	Meaning
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.
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 search through the quota file using wildcards. The two flags are: FIB$M_ALL_MEM Match all UIC members FIB$M_ALL_GRP Match all UIC groups The ACP maintains position context in FIB$L_WCC. On the first examine call, you specify 0 in FIB$L_WCC; the ACP returns a nonzero value so that each succeeding examine call returns the next matching entry. Read access to the quota file is required to examine all non-user entries.
FIB$C_MOD_QUOTA	Modify a disk quota file entry. The quota file entry specified by the UIC in the P2 argument block is modified according to the values in the block, as controlled by three flags in FIB$L_CNTRLVAL: FIB$M_MOD_PERM Change the permanent quota FIB$M_MOD_OVER Change the overdraft 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 Section 1.6.7.3). FIB$C_MOD_QUOTA requires write access to the quota file. The P3 and P4 arguments return the modified quota entry to you. By using the flags FIB$M_ALL_MEM and FIB$M_ALL_GRP, you can search through the quota file using wildcards just as you would with the FIB$C_EXA_QUOTA function.
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 P3 and P4 arguments return the removed quota file entry to you. By using the flags FIB$M_ALL_MEM and FIB$M_ALL_GRP, you can search through the quota file using wildcards just as you would with the FIB$C_EXAQUOTA function.

Figure 1–9 Quota File Transfer Block



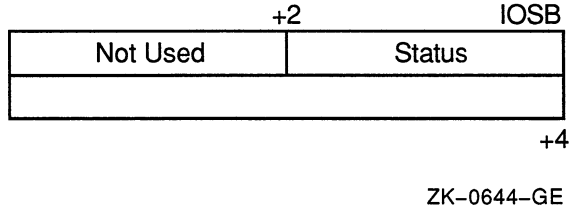
1.7 I/O Status Block

Figure 1–10 shows the I/O status block (IOSB) for ACP–QIO functions. Appendix A lists the status returns for these functions. (The *VMS System Messages and Recovery Procedures Reference Manual* provides explanations and suggested user actions for these returns.)

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, use the values returned in FIB locations.

Figure 1–10 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.

2 Card Reader Driver

This chapter describes the use of the VMS card reader driver that supports the CR11 card reader.

2.1 Supported Card Reader Device

The CR11 card reader reads standard 80-column punched data cards.

2.2 Driver Features

The VMS card reader driver provides the following features:

- Support for 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.

The VMS operating system provides the following card reader device- or function-dependent modifier bits for read data operations:

- IO$M_PACKED—Read packed Hollerith code
- IO$M_BINARY—Read binary code

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 special card punch combination that sets the translation mode (see Section 2.2.1.2), or by the set mode function (see Section 2.4.3).

2.2.1 Special Card Punch Combinations

The 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.

2.2.1.1 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 that punch combination.

2.2.1.2 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. (Table 2–5 lists the 026 and 029 punch codes.) 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.

The system can read cards that were punched on an 026 punch or an 029 punch. By default, the translation mode is 029; that is, the system reads cards from an 029 punch. However, you can change the translation mode by using the following:

- The SET CARD_READER command
- Translation mode cards

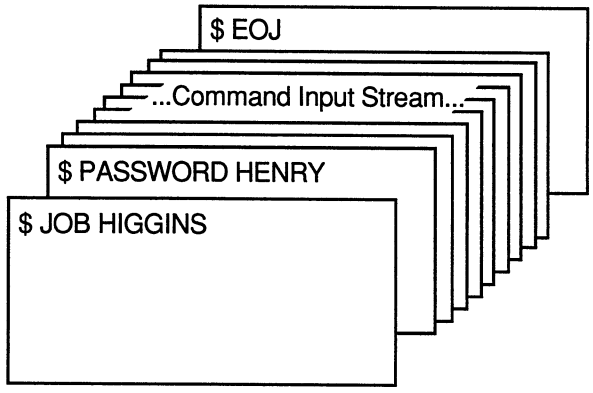
Use the SET CARD_READER command, with the /026 or /029 qualifier, to set the card reader to accept cards from either an 026 or an 029 card punch.

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

2.2.2 Submitting Batch Jobs Through the Card Reader

When you submit a batch job through a system card reader, precede the card deck containing the command procedure with cards containing JOB and PASSWORD commands. These cards specify your user name and password and, when executed, effect a login for you. The last card in the deck must contain the EOJ (End of Job) command. The EOJ card is equivalent to logging out. You can also use an overpunch card instead of an EOJ card to signal the end of a job. To do this, use an EOF card (12-11-0-1-6-7-8-9) overpunch or use the EOJ command. Figure 2–1 illustrates a card reader batch job.

Figure 2–1 A Card Reader Batch Job



ZK–0812–GE

When the system reads a job from the card reader, it validates the user name and password specified on the JOB and PASSWORD cards. Then, it copies the entire card deck into a temporary disk file named INPBATCH.COM in your default disk and directory, and it queues the job for batch execution. Thereafter, processing is the same as for jobs submitted interactively with the SUBMIT command. When the batch job is completed, the operating system deletes the INPBATCH.COM file.

You can prevent other users from seeing your password by suppressing printing when you keypunch the PASSWORD card.

2.2.3 Passing Data to Commands and Images

To pass data to commands and images in batch jobs that you submit through a card reader, you can do the following:

- Include the data in the command procedure by placing the data on the lines after the command or image that uses the data. Use the DECK and EOD commands if the data lines begin with dollar signs.
- Temporarily redefine SYS$INPUT as a file by using the DEFINE/USER_MODE command.

2.2.4 Error Recovery

The 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 every 30 seconds to the system operator.

- The current operation is retried every two seconds to test for a changed situation, such as 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.

When a recoverable card reader failure is detected and an error message is displayed on the system operator console, examine the card reader indicator lights to determine the reason for the failure. Any errors that occur must be fixed manually. The recovery is transparent to the user program issuing the I/O request.

The four categories of card reader failures and their respective recovery procedures are as follows:

- Pick check—The next card cannot be delivered from the input hopper to the read mechanism. To recover from this error, remove the next card to be read from the input hopper and smooth the leading edge (the edge that enters the read mechanism first). Replace the card in the input hopper and press the RESET button. The card reader operation resumes 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. To recover from this error, remove the last card read from the output hopper and examine it. If it is excessively worn or mutilated, repunch it. Place either card in the read station of the input hopper and press the RESET button. The card reader operation resumes 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. To recover from this error, 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. The card reader operation resumes automatically.
- Read check—The last card was read incorrectly. To recover from this error, remove the last card from the output hopper and examine it. If it is excessively worn, mutilated, or contains punches before column 0 or after column 80, repunch the card. Place either card in the read station of the input hopper and press the RESET button. The card reader operation resumes automatically. If the read check error recurs immediately, an adjustment is necessary.

2.3 Card Reader Driver Device Information

You can obtain information on card reader characteristics by using the Get Device/Volume Information ($GETDVI) system service. See the *VMS System Services Reference Manual.*

$GETDVI returns card reader characteristics when you specify the item codes DVI$_DEVCHAR and DVI$_DEVDEPEND. Tables 2–1 and 2–2 list these characteristics. The $DEVDEF macro defines the device-independent characteristics; the $CRDEF macro defines the device-dependent characteristics.

DVI$_DEVTYPE and DVI$_DEVCLASS return the device type and device class names, which are defined by the $DCDEF macro. The device class for card readers is DC$_CARD. The device type for the CR11 is DT$_CR11. DVI$_DEVBUFSIZ returns the buffer size. The default buffer size to be used for all card reader devices is 80 bytes.

Table 2–1 Card Reader Device-Independent Characteristics

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

[1] Defined by the $DEVDEF macro.

Table 2–2 Device-Dependent Characteristics for Card Readers

Value[1]	Meaning
CR$V_TMODE CR$S_TMODE	Specifies the translation mode for nonbinary, nonpacked Hollerith data transfers.[2] Possible values are: CR$K_T026 Translate according to 026 punch code CR$K_T029 Translate according to 029 punch code

[1] Defined by the $CRDEF macro.

[2] Section 2.2.1.2 describes the set translation mode punch code.

2.4 Card Reader Function Codes

The VMS card reader can perform logical, virtual, and physical I/O functions. Table 2–3 lists these functions and their function codes. These functions are described in more detail in the sections that follow.

Table 2–3 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

2.4.1 Read

Read is a function that 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.

The VMS operating system provides the following read function codes:

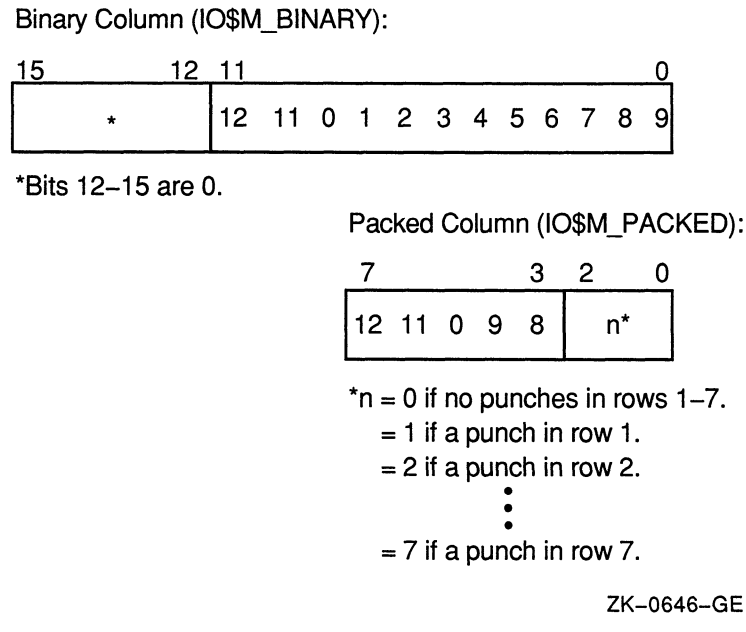
- IO$_READVBLK—Read virtual block
- IO$_READLBLK—Read logical block
- IO$_READPBLK—Read physical block

The following 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 are stored in sequential bytes of the input buffer. Figure 2–2 shows how data is stored by IO$M_BINARY and IO$M_PACKED.

Figure 2–2 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.

2.4.2 Sense Mode

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

2.4.3 Set Mode

Set mode operations affect the operation and characteristics of the associated card reader device. The VMS operating system defines the following types of set mode functions:

- Set mode
- Set characteristic

2.4.3.1 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. The following function code is provided.

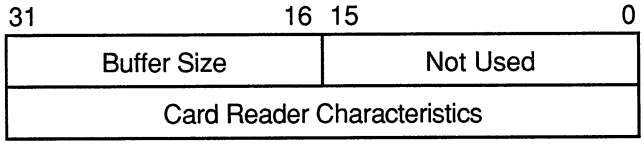
- IO$_SETMODE

This function takes the following device- or function-dependent argument:

- P1—The address of a characteristics buffer

Figure 2–3 shows the quadword set mode characteristics buffer.

Figure 2–3 Set Mode Characteristics Buffer



ZK–0647–GE

Table 2–4 lists the card reader characteristics and their meanings. The $CRDEF macro defines the characteristics values. Table 2–5 lists the 026 and 029 card reader codes.

Table 2–4 Set Mode and Set Characteristic Card Reader Characteristics

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 nor 029 mode is specified, the default mode can be set by the SET CARD_READER command.

Table 2–5 Card Reader Codes

Character	$ASCII_8$	DEC029	DEC026
{	173	12 0	12 0
}	175	11 0	11 0
SPACE	40	NONE	NONE

(continued on next page)

Table 2–5 (Cont.) Card Reader Codes

Character	$ASCII_8$	DEC029	DEC026
!	41	11 8 2	12 8 7
"	42	8 7	0 8 5
_	43	8 3	0 8 6
$	44	11 8 3	11 8 3
%	45	0 8 4	0 8 7
&	46	12	11 8 7
'	47	8 5	8 6
(	50	12 8 5	0 8 4
)	51	11 8 5	12 8 4
*	52	11 8 4	11 8 4
+	53	12 8 6	12
,	54	0 8 3	0 8 3
-	55	11	11
.	56	12 8 3	12 8 3
/	57	0 1	0 1
0	60	0	0
1	61	1	1
2	62	2	2
3	63	3	3
4	64	4	4
5	65	5	5
6	66	6	6
7	67	7	7
8	70	8	8
9	71	9	9
:	72	8 2	11 8 2
;	73	11 8 6	0 8 2
<	74	12 8 4	12 8 6
=	75	8 6	8 3
>	76	0 8 6	11 8 6
?	77	0 8 7	12 8 2
@	100	8 4	8 4
A	101	12 1	12 1
B	102	12 2	12 2
C	103	12 3	12 3
D	104	12 4	12 4

(continued on next page)

Table 2–5 (Cont.) Card Reader Codes

Character	$ASCII_8$	DEC029	DEC026
E	105	12 5	12 5
F	106	12 6	12 6
G	107	12 7	12 7
H	110	12 8	12 8
I	111	12 9	12 9
J	112	11 1	11 1
K	113	11 2	11 2
L	114	11 3	11 3
M	115	11 4	11 4
N	116	11 5	11 5
O	117	11 6	11 6
P	120	11 7	11 7
Q	121	11 8	11 8
R	122	11 9	11 9
S	123	0 2	0 2
T	124	0 3	0 3
U	125	0 4	0 4
V	126	0 5	0 5
W	127	0 6	0 6
X	130	0 7	0 7
Y	131	0 8	0 8
Z	132	0 9	0 9
[	133	12 8 2	11 8 5
\	134	11 8 7	8 7
]	135	0 8 2	12 8 5
↑ or ^	136	12 8 7	8 5
← or _	137	0 8 5	8 2

Application programs that change specific card reader characteristics should first use the IO$_SENSEMODE function to read the current characteristics, modify them, and then use the set mode function to write back the results. Failure to follow this sequence results in clearing any previously set characteristic.

2.4.3.2 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. The following function code is provided:

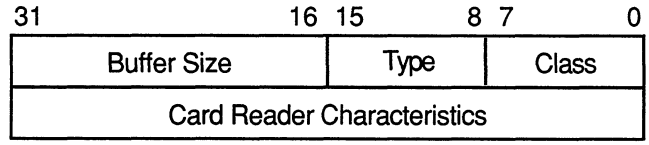
- IO$_SETCHAR

This function takes the following device- or function-dependent argument:

- P1—The address of a characteristics buffer

Figure 2–4 shows the set characteristic characteristics buffer.

Figure 2–4 Set Characteristic Buffer



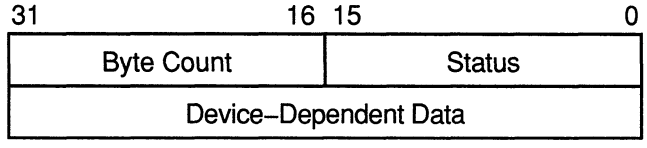
ZK–0648–GE

The device type value is DT$_CR11. The device class value is DC$_CARD. Table 2–4 lists the card reader characteristics for the Set Characteristic function.

2.5 I/O Status Block

The I/O status block (IOSB) format for QIO functions on the card reader is shown in Figure 2–5. Appendix A lists the status returns for these functions. (The *VMS System Messages and Recovery Procedures Reference Manual* provides explanations and suggested user actions for these returns.) Table 2–2 lists the device-dependent data returned in the second longword. The IO$_SENSEMODE function can be used to obtain this data.

Figure 2–5 IOSB Contents



ZK–0649–GE

3 Disk Drivers

This chapter describes the use of VMS disk drivers. These drivers support the devices listed in Table 3–1.

All disk drivers support Files–11 On-Disk Structure Level 1 and Level 2 file structures. Access to these file structures is through the DCL commands INITIALIZE and MOUNT, followed by the VMS RMS calls described in the *VMS Record Management Services Manual*. Files in RT–11 format can be read or written with the file exchange facility EXCHANGE.

3.1 Supported Disk Devices and Controllers

The following sections provide greater detail about the disk devices listed in Table 3–1. To obtain additional information about a device, use the DCL command SHOW DEVICE with the /FULL qualifier, the Get Device/Volume Information ($GETDVI) system service (from a program), or the F$GETDVI lexical function (in a command line or command procedure). Section 3.3 lists the information on disk devices returned by $GETDVI.

Table 3–1 Supported Disk Devices

Disk Device	Code	Type	DSA	Capacity (Logical Blocks/Drive)
RA60	DJ	Removable	Yes	400,176
RA70	DU	Fixed	Yes	547,041
RA80	DU	Fixed	Yes	236,964
RA81	DU	Fixed	Yes	891,072
RA82	DU	Fixed	Yes	1,216,665
RA90	DU	Fixed	Yes	2,343,750
RB02	DQ	Cartridge	No	20,480
RB80	DQ	Fixed	No	242,606
RC25	DA	Fixed, Cartridge	Yes[1]	102,400[2]
RD32	DU	Fixed	Yes[1,3]	83,204
RD51	DU	Fixed	Yes[1,3]	21,600

[1] Incompatible with the UDA50, KDA50, KDB50, HSC40, HSC50, and HSC70 disk controllers.

[2] 51,200 fixed; 51,200 cartridge.

[3] The RD series of disk drives conforms to DSA when used with the RQDX series of controllers. RD-series disk drives do not conform to DSA when used on a VAXstation 2000.

(continued on next page)

Disk Drivers

3.1 Supported Disk Devices and Controllers

Table 3–1 (Cont.) Supported Disk Devices

Disk Device	Code	Type	DSA	Capacity (Logical Blocks/Drive)
RD52	DU	Fixed	Yes[1,3]	60,480
RD53	DU	Fixed	Yes[1,3]	138,672
RD54	DU	Fixed	Yes[1,3]	311,200
RF30	DI	Fixed	Yes[1]	292,968
RF71	DI	Fixed	Yes[1]	781,250
RL02	DL	Cartridge	No	20,480
RM03	DR	Removable	No	131,680
RM05	DR	Removable	No	500,384
RM80	DR	Fixed	No	242,606
RP05	DB	Removable	No	171,798
RP06	DB	Removable	No	340,670
RP07	DR	Fixed	No	1,008,000
RK06	DM	Cartridge	No	27,126
RK07	DM	Cartridge	No	53,790
RRD40	DU or DK[4]	Optical Optical	Yes[1] No	1,669,400 1,669,400
RRD50	DU	Optical	Yes[1]	1,669,400
RX01	DX	Flexible	No	494
RX02	DY	Flexible	No	494[5] 988 [6]
RX23	DU	Flexible	Yes[1]	2,734
RX33	DU	Flexible	Yes[1]	2,400
RX50	DU	Flexible	Yes[1]	800
RZ22	DK	Fixed	No	101,563
RZ23	DK	Fixed	No	203,125
RZ55	DK	Fixed	No	742,188
TU58[7]	DD	Cartridge	No	512

[1] Incompatible with the UDA50, KDA50, KDB50, HSC40, HSC50, and HSC70 disk controllers.

[3] The RD series of disk drives conforms to DSA when used with the RQDX series of controllers. RD-series disk drives do not conform to DSA when used on a VAXstation 2000.

[4] SCSI interface RRD40 compact disc drive.

[5] Single density (See Section 3.3).

[6] Double density (See Section 3.3).

[7] A magnetic tape device, the TU58 operationally resembles a disk device. See Section 3.1.24 for a description of the TU58 in disk terms.

3.1.1 UDA50 UNIBUS Disk Adapter

The UDA50 UNIBUS Disk Adapter (UDA50) is a microprocessor-based disk controller for mass storage devices that implement the DIGITAL Storage Architecture (DSA); for more information on the DSA, see Section 3.2.3.

The UDA50 is used to connect any combination of four RA60, RA80, and RA81 disk drives to the UNIBUS. Two UDA50 controllers can be attached to a single UNIBUS for a maximum of eight disk drives per UNIBUS. On the VAX–11/780 processor, the VMS operating system supports one UDA50 on the first UNIBUS, which can accommodate certain other options. Adding a second UDA50 requires a second UNIBUS. With the exception of the first UNIBUS, a maximum of two UDA50s per UNIBUS are supported. If two UDA50s are on a UNIBUS, no other options can be placed on that UNIBUS. The VAX–11/730 processor supports only one UDA50 per UNIBUS.

The UDA50, in implementing DSA, takes over the control of the physical disk unit. The VMS operating system processes request virtual or logical I/O on disks controlled by the UDA50. The VMS operating system maps virtual block addresses into logical block addresses. The UDA50 then resolves logical block addresses into physical block addresses on the disk.

The UDA50 corrects bad blocks on the disk by requesting that the disk class driver **revector** a failing physical block to another, error-free physical block on the disk; the logical block number is not changed (see Section 3.2.10.1). Any bad blocks that might exist on a disk attached to a UDA50 are transparent to the VMS operating system, which does logical or virtual I/O to such a disk. The UDA50 also corrects most data errors.

3.1.2 KDA50 Disk Controller

The KDA50 disk controller is a two-module disk controller that allows the RA-series DSA disk drives to be attached to Q-bus systems. The KDA50 performs the same functions as the UDA50 (see Section 3.1.1).

3.1.3 KDB50 Disk Controller

The KDB50 disk controller is a two-module disk controller that allows the RA-series DSA disk drives to be attached to BI bus systems, such as the VAX 8200 processor. The KDB50 performs the same functions as the UDA50 (see Section 3.1.1).

3.1.4 HSC-Series Controllers

The HSC series of intelligent disk controllers consists of the HSC40, HSC50, and the HSC70. HSC controllers are high-speed, high-availability controllers for mass storage devices that implement the DIGITAL Storage Architecture (DSA); for more information about the DSA, see Section 3.2.3. An HSC controller is connected to a processor by a Computer Interconnect

(CI). The VMS operating system supports the use of the HSC40, HSC50, HSC70 in controlling the RA family of disks.

The HSC40 can support up to 12 SDI disks from the SA or RA families of disk drives or a combination of up to 12 SDI disk drives and TA-series tape drives.

The HSC70 can support up to 32 SDI disks from the SA or RA families of disk drives or a combination of SDI disk drives and TA-series tape drives.

HSC controllers, in implementing DSA, take over the control of the physical disk unit. VMS operating system processes request virtual or logical I/O on disks controlled by the HSC controller. The VMS operating system maps virtual block addresses into logical block addresses. The HSC controller then resolves logical block addresses into physical block addresses on the disk.

HSC controllers correct bad blocks on the disk by revectoring a failing physical block to another, error-free physical block on the disk; the logical block number is not changed. The VMS operating system, which performs logical or virtual I/O to such a disk, does not recognize that any bad blocks might exist on a disk attached to an HSC controller. HSC controllers also correct most data errors.

The HSC series of controllers provides access to disks despite most hardware failures. Use of an HSC controller permits two or more processors to access files on the same disk.

Note: Only one system should have write access to a Files–11 On-Disk Structure Level 1 disk or to a foreign-mounted disk; all other systems should only have read access to the disk. For Files–11 On-Disk Structure Level 2 volumes, the VMS operating system enables read/write access to all nodes that are members of the same VAXcluster.

HSC-series controllers allow you to add or subtract disks from the device configuration without rebooting the system.

3.1.5 SII Integral Adapter

The SII integral adapter on the MicroVAX 3300/3400 provides access through the DIGITAL Small Storage Interconnect (DSSI) bus to a maximum of seven storage devices.

The term **dual-host** refers to pairs of CPUs connected to a bus. In dual-host configurations of pairs of MicroVAX 3300/3400 CPUs, the DSSI bus must be connected between the SII integral adapters present on both CPUs.

A maximum of six devices can be connected to the SII integral adapter in dual-host configurations.

3.1.6 KFQSA Adapter

The KFQSA adapter allows a maximum of seven storage devices for use on Q-bus systems.

In dual-host configurations of pairs of MicroVAX 3800/3900 CPUs, the DSSI bus must be connected between KFQSA adapters present on both CPUs.

A maximum of six devices can be connected to the KFQSA adapter in dual-host configurations.

3.1.7 RQDX3 Controller

The RQDX3 is a Q-bus controller used with the RD series of Winchester-type disk drives and the RX33 and RX50 flexible diskette drives.

3.1.8 RA70 and RA90 Disk Drives

The RA70 is a 5.25-inch 280-megabyte (MB) high-performance DSA disk drive that uses thin-film media. It has an average access time of 27.0 milliseconds (ms) and average seek time of 19.5 ms. The RA70 uses the Standard Disk Interconnect (SDI) and the KDA50 controller, and can be dual-ported.

The RA90 is a 1.2-gigabyte disk drive designed with thin-film heads and 9-inch thin-film media with an average seek time of 18.5 ms. The RA90 conforms to DSA and uses the SDI. Both the RA70 and RA90 disk drives can be connected to medium-sized systems with the HSC-series controllers, KDB50, or UDA50 controllers.

3.1.9 RA60 Disk

The RA60 device uses high-capacity, removable media that provides 205 MB of usable storage (7.5 million bits of data per square inch) with transfer rates of 1.9 MB per second (burst) and 950 KB per second (sustained). The RA60 belongs to the DIGITAL Storage Architecture (DSA) family of disk devices (see Section 3.2.3). It is connected to either a UNIBUS Disk Adapter (UDA50) or an HSC50 controller. Up to four disk drives can be connected to each UDA50. Up to 24 disk drives can be connected to each HSC50.

3.1.10 RA80/RB80/RM80 and RA81 Fixed-Media Disks

The R80 disk drive is a high-capacity, moving-head disk whose nonremovable media consists of 14 data surfaces. Depending on how it is connected to the system, the R80 is identified internally as an RA80, RB80, or RM80, as follows:

- RA80—An R80 connected to the system through a UNIBUS disk adapter (UDA50) or an HSC50 controller. Up to four disk drives can

be connected to each UDA50. Up to 24 disk drives can be connected to each HSC50.

- RB80—An R80 connected to the system through an RB730 controller on a VAX 11/730 processor. Of the maximum of four drives that can be connected to an RB730 controller, only one can be an RB80.
- RM80—An R80 connected to the system through a MASSBUS adapter (MBA). Up to eight disk drives can be connected to each MBA.

The RA81 is a high-capacity disk drive with nonremovable media that can hold more than 890,000 blocks of data. This translates into more than 455 MB per spindle. The RA81 is connected to a UDA50 or an HSC50 controller. Up to four disk drives can be connected to each UDA50. Up to 24 drives can be connected to each HSC50.

The RA80 and RA81 belong to the DIGITAL Storage Architecture (DSA) family of disk devices (see Section 3.2.3).

3.1.11 RB02 and RL02 Cartridge Disk

The RL02 cartridge disk is a removable, random-access mass storage device with two data surfaces. The RL02 is connected to the system by an RL11 controller that interfaces with the UNIBUS adapter. Up to four RL02 disk drives can be connected to each RL11 controller. For physical I/O transfers, the track, sector, and cylinder parameters describe a physical 256-byte RL02 sector (see Section 3.4).

When the RL02 is connected to an RB730 controller on a VAX–11/730 processor, it is identified internally as an RB02 disk drive. Disk geometry is unchanged and RL02 disk packs can be exchanged between drives on different controllers. Up to four drives can be connected to the RB730 controller.

3.1.12 RC25 Disk

The RC25 disk is a self-contained, Winchester-type, mass storage device that consists of a disk adapter module, a disk drive, and an integrated disk controller. The drive contains two 8-inch, double-sided disks. One of the disks (RCF25) is a sealed, nonremovable, fixed-media disk. The other disk is a removable cartridge disk that is sealed until it is loaded into the disk drive. The disks share a common drive spindle, and together they provide 52 million bytes of storage. Adapter modules interface the RC25 with either a UNIBUS system or a Q-bus system.

3.1.13 RD-Series Disks

The RD53 and RD54 are 5.25-inch, full-height, Winchester-type drives with average access time of 38 ms and a data transfer rate of 0.625 MB per second. The RD53 and RD54 have a formatted capacity of 71 MB and 159 MB, respectively. When used with the RQDX3 controller, the RD53 and RD54 are DSA disks.

See Section 3.2.11 for information about using RD series disks on the VAXstation 2000.

3.1.14 RF-Series Disks

The RF series of Winchester-type disk drives consists of the RF30 and the RF71. The RF30 is a 150-MB, 5.25-inch, half-height disk drive while the RF71 is a 400-MB full-height disk drive. The RF30 and RF71 include an embedded controller for multihost access and a Mass Storage Communications Protocol (MSCP) server. The RF71 has a peak data transfer rate of 1.5 MB per second with average seek and access time of 21 ms and 29 ms, respectively.

Both the RF30 and RF71 disks use DIGITAL Storage System Interconnect (DSSI) bus and host adapters.

3.1.15 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 that interfaces to the UNIBUS adapter. Up to eight disk drives can be connected to each RK611.

3.1.16 RM03 and RM05 Pack Disks

The RM03 and RM05 pack disks are removable, moving-head disks that consist of five data surfaces for the RM03 and 19 data surfaces for the RM05. These disks are connected to the system by a MASSBUS adapter (MBA). Up to eight disk drives can be connected to each MBA.

3.1.17 RP05 and RP06 Disk

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

3.1.18 RP07 Fixed Media Disk

The RP07 is a 516-MB, fixed media disk drive that attaches to the MASSBUS of the VAX–11/780 system. The RP07 transfers data at 1.3 million bytes per second or as an option at a peak rate of 2.2 million bytes per second. The nine platters rotate at 3600 rpm and their data is accessed at an average speed of 31.3 milliseconds. These disks are connected to the system by an MBA. Up to eight disk drives can be connected to each MBA.

3.1.19 RRD40 and RRD50 Read-Only Memory (CDROM)

The RRD40 and RRD50 are Compact Disc Read-Only Memory (CDROM) devices that use replicated media with a formatted capacity of approximately 600 MB.

The RRD40 is a 5.25-inch half-height, front-loading table-top or embedded device that attaches to the system using either the Small Computer System Interface (SCSI) or Q-bus interface.

The RRD50 is a 5.25-inch, top-loading table-top device that attaches to the system using a Q-bus interface.

The RRD40 has an average access time of 0.5 seconds while the average access time for the RRD50 is 1.5 seconds. Both the RRD40 and RRD50 have a data transfer rate of 150 KB per second.

The media for the RRD40 and the RRD50 are removable 4.7-inch (120 mm) compact disks. However, the media for the RRD40 are enclosed in protective self-loading carriers. The RRD40 with a SCSI interface is also available as an embedded unit. The RRD40 and RRD50 Q-bus subsystems are standard disk MSCP devices.

3.1.20 RX01 Console Disk

The RX01 disk uses a diskette. 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 logical and virtual block I/O operations, data is accessed with one block resolution (four sectors). The sector numbers are interleaved to expedite data transfers. Section 3.2.9 describes sector interleaving in greater detail.

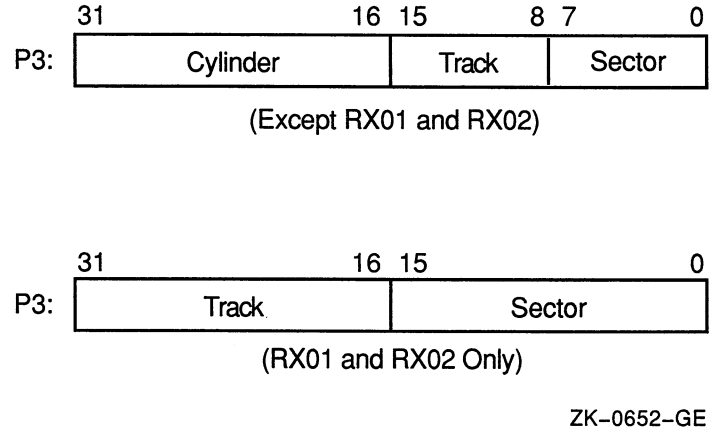
For physical block I/O operations, the track, sector, and cylinder parameters describe a physical, 128-byte RX01 sector (see Figure 3–1 and Section 3.4). Note that the driver does not apply track-to-track skew, cylinder offset, or sector interleaving to this physical medium address.

3.1.21 RX02 Disk

The RX02 disk is a mass storage device that uses removable diskettes. The RX02 supports existing single-density, RX01-compatible diskettes. A double-density mode allows diskettes to be recorded at twice the linear density. An entire diskette must be formatted in either single or double density. Mixed mode diskettes are not allowed.

The RX02 is connected to the system by an RX211 controller that interfaces with the UNIBUS adapter. Up to two disk drives can be controlled by each RX211.

Figure 3–1 Disk Physical Address



For logical and virtual block I/O operations, data is accessed with single block resolution (four single-density sectors or two double-density sectors). The sector numbers are interleaved to expedite data transfers. Section 3.2.9 describes this feature in greater detail.

For physical block I/O operations, the track and sector parameters shown in Figure 3–1 describe a physical sector (128 bytes in single density; 256 bytes in double density). The driver does not apply track-to-track skew, cylinder offset, or sector interleaving to the physical medium address.

3.1.22 RX-Series Drives

The following sections describe the RX family of flexible diskette drives.

3.1.22.1 RX23

The RX23 device is a one-inch high, flexible diskette drive that uses 3.5-inch microfloppy diskettes. The RX23 drive can access standard- and high-density media. The following table summarizes capacities for standard- and high-density media.

Density	Unformatted	Formatted
Standard	1.0 MB	700 KB
High	2.0 MB	1.4 MB

The RX23 is backwardly compatible in that it can read 1-MB media. It can also read and write 2.0-MB double-sided, high-density (135 tracks per inch) media.

The RX23 communicates with the controller using the ST506 fixed disk interconnect (FDI).

3.1.22.2 RX33

The RX33 is a 1.2 MB, 5.25-inch, half-height diskette drive. The RX33 can record in either standard- or high-density mode. High-density mode provides 1.2 MB of storage using 96 tracks per inch using double-sided, high-density diskettes.

In standard-density mode, the RX33 drive is read- and write-compatible with single-sided, standard-density RX50 diskettes.

3.1.22.3 RX50

The RX50 dual diskette drive stores data in fixed-length blocks on 5.25-inch 0.8-MB, flexible diskettes using preformatted headers. The RX50 can accommodate two diskettes simultaneously.

3.1.23 RZ-Series Disks

The RZ series of Winchester-type disk drives consists of the RZ22, RZ23, and the RZ55. The RZ22 and RZ23 are 3.5-inch, half-height SCSI drives with an average seek rate of 33 ms and an average data transfer rate of 1.25 MB per second. The RZ22 and RZ23 have a capacity of 52 MB and 104 MB, respectively.

The RZ55 is a 332-MB, 5.25-inch, full-height SCSI drive with an average access rate of 24 ms.

3.1.24 TU58 Magnetic Tape (DECtape II)

The TU58 is a random-access, mass storage magnetic tape device capable of reading and writing 256K bytes per drive of data on block-addressable, preformatted cartridges at 800 bits per inch. Unlike conventional magnetic tape systems, which store information at variable positions on the tape, the TU58 stores information at fixed positions on the tape, as do magnetic disk or floppy disk devices. Thus, blocks of data can be placed on tape in a random fashion, without disturbing previously recorded data. In its physical geometry, the tape is conceptually viewed as having one cylinder, four tracks per cylinder, and 128 sectors per track. Each sector contains one 512-byte block.

The TU58 uses two vectors. NUMVEC=2 is required on the CONNECT command when specifying SYSGEN parameters.

The TU58 interfaces with the UNIBUS adapter through a DL11-series interface device. Both the TU58 and the DL11 should be set to 9600 baud. (Because the TU58 is attached to a DL11, the user cannot directly access the TU58 registers if the TU58 is on the UNIBUS.) The *DIGITAL Terminals and Communications Handbook* provides additional information on the DL11. The TU58, which has its own controller, can access either one or two tape drives.

3.2 Driver Features

VMS disk drivers provide the following features:

- Multiple controllers of the same type (except RB730), for example, more than one MBA or RK611 can be used on the system
- Multiple disk drives per controller (The exact number depends on the controller.)
- Different types of disk drives on a single controller
- Static dual porting (MBA drives only)
- Overlapped seeks (except RL02, RX01, RX02, and TU58)
- Data checks on a per-request, per-file, or per-volume basis (except RX01 and RX02)
- Full recovery from power failure for online disk drives with volumes mounted
- Extensive error recovery algorithms, such as error code correction and offset (except RB02, RL02, RX01, RX02, and TU58); for DSA disks, these algorithms are implemented in the controller
- 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
- Logical-to-physical sector translation (RX01 and RX02 only)

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

3.2.1 Dual-Pathed Disks

A **dual-pathed disk** is a dual-ported disk that is accessible to all the CPUs in the VAXcluster, not just to the CPUs that are connected physically to the disk. Dual-pathed disks can be any of the following:

- Dual-ported MASSBUS disks
- Dual-ported HSC disks

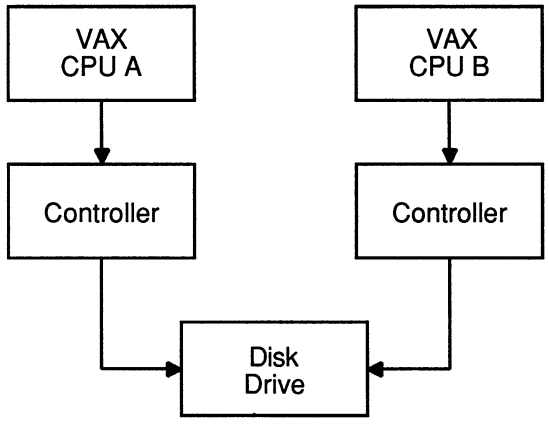
- Dual-pathed DSA disks on local UDA50, KDA50, and KDB50 controllers
- Dual-ported RF-series disks

The term dual-pathed refers to the two paths through which clustered CPUs can access a disk to which they are not directly connected. If one path fails, the disk is accessed over the other path. (Note that with a dual-ported MASSBUS disk, a CPU directly connected to the disk always accesses it locally.)

3.2.2 Dual Porting MASSBUS Disks

The VMS MASSBUS disk drivers, DBDRIVER and DRDRIVER, support static dual porting. Dual porting allows two MASSBUS controllers to access the same disk drive. Figure 3–2 shows this configuration. The RP05, RP06, RP07, RM03, RM05, and RM80 disk drives can be ordered, or upgraded in the field, with the MASSBUS dual port option.

Figure 3–2 Dual-Ported Disk Drives



ZK–0650–GE

3.2.2.1 Port Selection and Access Modes

The port select switches, on each disk drive, select the ports from which the drive can be accessed. A drive can be in one of the following access modes:

- Locked on Port A—The drive is in a single-port mode (Port A). It does not respond to any request on Port B.
- Locked on Port B—The drive is in a single-port mode (Port B). It does not respond to any request on Port A.

- Programmable A/B—The drive is capable of responding to requests on either Port A or Port B. In this mode, the drive is always in one of the following states:
  - — The drive is connected and responding to a request on Port A. It is closed to requests on Port B.
  - — The drive is connected and responding to a request on Port B. It is closed to requests on Port A.
  - — The drive is in a neutral state. It is equally available to requests on either port on a first-come, first-serve basis.

The operational condition of the drive cannot be changed with the port select switches after the drive becomes ready. To change from one mode to another, the drive must be in a nonrotating condition. After the new mode selection has been made, the drive must be restarted.

If a drive is in the neutral state and a disk controller either reads or writes to a drive register, the drive immediately connects a port to the requesting controller. For read operations, the drive remains connected for the duration of the operation. For write operations, the drive remains connected until a release command is issued by the device driver or a one second timeout occurs. After the connected port is released from its controller, the drive checks the other port's request flag to determine whether there has been a request on that port. If no request is pending, the drive returns to the neutral state.

3.2.2.2 Disk Use and Restrictions

If the volume is mounted foreign, read/write operations can be performed at both ports provided the user maintains control of where information is stored on the disk.

The Autoconfigure Utility currently may not be able to locate the nonactive port. For example, if a dual-ported disk drive is connected and responding at Port A, the CPU attached to Port B might not be able to find Port B with the Autoconfigure Utility. If this problem occurs, execute the AUTOCONFIGURE ALL/LOG command after the system is running.

3.2.2.3 Restriction on Dual-Ported Non-DSA Disks in a VAXcluster

Do not use SYSGEN to AUTOCONFIGURE or CONFIGURE a dual-ported, non-DSA disk that is already available on the system through use of an MSCP server. Establishing a local connection to the disk when a remote path is already known creates two uncoordinated paths to the same disk. Use of these two paths may corrupt files and data on any volume mounted on the drive.

Note: If the disk is not dual-ported or is never served by an MSCP server on the remote host, this restriction does not apply.

In a VAXcluster, dual-ported non-DSA disks (MASSBUS or UNIBUS) can be connected between two nodes of the cluster. These disks can also be made available to the rest of the cluster using the MSCP server on either or both of the hosts to which a disk is connected.

If the local path to the disk is not found during the bootstrap, then the MSCP server path from the other host will be the only available access to the drive. The local path will not be found during a boot if any of the following conditions exist:

- The port select switch for the drive is not enabled for this host.
- The disk, cable, or adapter hardware for the local path is broken.
- There is sufficient activity on the other port to hide the existence of the port.
- The system is booted in such a way that the SYSGEN AUTOCONFIGURE ALL command in the SYS$SYSTEM:STARTUP.COM procedure was not executed.

Use of the disk is still possible through the MSCP server path.

After the configuration of the disk has reached this state, it is important **not** to add the local path back into the system I/O database. Because the VMS operating system does not provide an automatic method for adding this local path, the only possible way that you can add this local path is to use the Sysgen Utility (SYSGEN) qualifiers AUTOCONFIGURE or CONFIGURE to configure the device. SYSGEN is currently not able to detect the presence of the disk's MSCP path, and will incorrectly build a second set of data structures to describe it. Subsequent events could lead to incompatible and uncoordinated file operations, which might corrupt the volume.

To recover the local path to the disk, it is necessary to reboot the system connected to that local path.

See the *VMS VAXcluster Manual* for additional information on dual-ported disk operation.

3.2.3 Dual-Pathed DSA Disks

A dual-ported DSA disk can be failed over between the two CPUs that serve it to the VAXcluster under the following conditions: (1) the same disk controller letter and allocation class are specified on both CPUs and (2) both CPUs are running the MSCP server.

Note: Failure to observe these requirements can endanger data integrity.

However, because a DSA disk can be online to only one controller at a time, only one of the CPUs can use its local connection to the disk. The second CPU accesses the disk through the MSCP server. If the CPU that is currently serving the disk fails, the other CPU detects the failure and fails the disk over to its local connection. The disk is thereby made available to the VAXcluster once more.

Note: A dual-ported DSA disk may not be used as a system disk.

3.2.4 Dual-Porting HSC Disks

By design, HSC disks are cluster accessible. Therefore, if they are dual ported, they are automatically dual pathed. CI-connected CPUs can access a dual-pathed HSC disk by way of a path through either HSC-connected device.

For each dual-ported HSC disk, you can control failover to a specific port using the port select buttons on the front of each drive. By pressing either port select button (A or B) on a particular drive, you can cause the device fail over to the specified port.

With the port select button, you can select alternate ports to balance the disk controller workload between two HSC subsystems. For example, you could set half of your disks to use port A and set the other half to use port B.

The port select buttons also allow you to fail over all the disks to an alternate port manually when you anticipate the shutdown of one of the HSC subsystems.

3.2.5 Dual-Pathed RF-Series Disks

In a dual-path configuration of pairs of MicroVAX 3300/3400 CPUs or pairs of MicroVAX 3800/3900 CPUs using RF-series disks, CPUs have concurrent access to any disk on the DSSI bus. A single disk is accessed through two paths and can be served to all satellites by either CPU.

If either CPU fails, satellites can access their disks through the remaining CPU. Note that failover occurs in the following situations: (1) when the DSSI bus is connected between SII integral adapters on both MicroVAX 3300/3400 CPUs or (2) when the DSSI bus is connected between the KFQSA adapters on pairs of MicroVAX 3300/3400s or pairs of MicroVAX 3800/3900s.

Note: The DSSI bus should not be connected between a KFQSA adapter on one CPU and an SII integral adapter on another.

3.2.6 Data Check

A data check is made after successful completion of a read or write operation and, except for the TU58, compares the data in memory with the data on disk to make sure they match.

Disk drivers support data checks at the following levels:

- Per request—You can specify the data check function modifier (IO$M_DATACHECK) on a read logical block, write logical block, read virtual block, write virtual block, read physical block, or write physical block operation. IO$M_DATACHECK is not supported for the RX01 and RX01 drivers.

- Per volume—You can specify the characteristics "data check all reads" and "data check all writes" when the volume is mounted. The *VMS DCL Dictionary* describes volume mounting and dismounting. The *VMS System Services Reference Manual* describes the Mount Volume ($MOUNT) and Dismount Volume ($DISMOU) system services.
- Per file—You can specify the file access attributes "data check on read" and "data check on write." File access attributes are specified when the file is accessed. Chapter 1 of this manual and the *VMS Record Management Services Manual* describe file access.

Offset recovery is performed during a data check but Error Code Correctable (ECC) correction is not performed (see Section 3.2.8). For example, if a read operation is performed and an ECC correction is 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 the data check could not be performed because of an ECC correction.

Data checks on read operations are extremely rare, and you 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.

A data check operation directed to a TU58 does not compare the data in memory with the data on tape. Instead, either a read check or a write check operation is performed (see Sections 3.4.1 and 3.4.2).

3.2.7 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). Seek operations can be overlapped **except** on RL02, RX01, RX02, TU58 drives, MicroVAX 2000, VAXstation 2000, or on controllers with floppy disks (for example, RQDX3) when the disk is executing I/O requests. 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.

All DSA controllers perform extensive seek optimization functions as part of their operation; IO$M_INHSEEK has no effect on these controllers.

3.2.8 Error Recovery

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

- Handling special conditions such as power failure and interrupt timeout.
- Retrying nonfatal controller and drive errors. For DSA and SCSI disks, this function is implemented by the controller.
- Applying error correction information (not applicable for RB02, RL02, RX01, RX02, and TU58). For DSA and SCSI disks, error correction is implemented by the controller.
- Offsetting read heads to try to obtain a stronger recorded signal (not applicable for RB02, RL02, RB80, RM80, RX01, RX02, and TU58). For DSA and SCSI disks, this function is implemented by the controller.

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 online, 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 executed up to eight times before a fatal error is returned.

All normal error recovery procedures (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.

3.2.8.1 Skip Sectoring

Skip sectoring is a bad block treatment technique implemented on R80 disk drives (the RB80 and RM80 drives). In each track of 32 sectors, one sector is reserved for bad block replacement. Consequently, an R80 drive has available only 31 sectors per track. The Get Device/Volume Information ($GETDVI) system service returns this value.

You can detect bad blocks when a disk is formatted. Most formatters place these blocks in a bad block file. On an R80 drive, the first bad block encountered on a track is designated as a skip sector. This is accomplished by setting a flag in the sector header on the disk and placing the block in the skip sector file.

When a skip sector is encountered during a data transfer, it is skipped over, and all remaining blocks in the track are shifted by one physical block. For example, if block number 10 is a skip sector, and a transfer request was made beginning at block 8 for four blocks, then blocks 8, 9, 11, and 12 will be transferred. Block 10 will be "skipped."

Because skip sectors are implemented at the device driver level, they are not visible to you. The device appears to have 31 contiguous sectors per track. Sector 32 is not directly addressable, although it is accessed if a skip sector is present on the track.

3.2.9 Logical-to-Physical Translation (RX01 and RX02)

Logical block-to-physical sector translation on RX01 and RX02 drives adheres to the standard VMS format. For each 512-byte logical block selected, the driver reads or writes four 128-byte physical sectors (or two 256-byte physical sectors if an RX02 is in double-density mode). To minimize rotational latency, the physical sectors are interleaved. Interleaving 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 medium 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 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-25
ISECT = ITEMP
```

Skew:

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

3 Set the sector number in the range of 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
```

3.2.10 DIGITAL Storage Architecture (DSA) Devices

The DIGITAL Storage Architecture (DSA) is a collection of specifications that cover all aspects of a mass storage product. The specifications are grouped into the following general categories:

- Media format—Describes the structure of sectors on a disk and the algorithms for replacing bad blocks
- Drive-to-controller interconnect—Describes the connection between a drive and its controller
- Controller-to-host communications—Describes how hosts request controllers to transfer data

Because the VMS operating system supports all DSA disks, it supports all controller-to-host aspects of DSA. Some of these disks, such as the RA60, RA80, and RA81, use the standard drive-to-controller specifications. Other disks, such as the RC25, RD51, RD52, RD53, and RX50, do not. Disk systems that use the standard drive-to-controller specifications employ the same hardware connections and use the HSC50, KDA50, KDB50, and UDA50 interchangeably. Disk systems that do not use the drive-to-controller specifications provide their own internal controller, which conforms to the controller-to-host specifications.

DSA disks differ from MASSBUS and UNIBUS disks in the following ways:

- DSA disks contain no bad blocks. The hardware and the disk class driver (DUDRIVER) function to ensure a logically contiguous range of good blocks. If any block in the user area of the disk develops a defective area, all further access to that block is revectored to a spare good block. Consequently, it is never necessary to run the Bad Block Locator Utility (BAD) on DSA disks. There is no manufacturer's bad block list and the file BADBLK.SYS is empty. (The Verify Utility, which is invoked by the ANALYZE /DISK_STRUCTURE /READ_CHECK command, can be used to check the integrity of newly received disks.) See Section 3.2.10.1 for additional information about bad block replacement for DSA disks.
- Insert a WAIT statement in your SYSTARTUP_V5.COM file prior to the first MOUNT statement for a DSA disk. The wait period should be about two to four seconds for the UDA50 and about 30 seconds for the HSC50. The wait time is controller-dependent and can be as much as several minutes if the controller is offline or otherwise inoperative. If this wait is omitted, the MOUNT request may fail with a "no such device" status.
- The DUDRIVER and the DSA device controllers allow multiple, concurrently outstanding QIO requests. The order in which these requests complete might not be in the order in which they were issued.

- All DSA disks can be dual-ported, but only one HSC/UDA controller can control a disk at a time (see Section 3.2.3).
- In many cases, you can attach a DSA disk to its controller on a running VMS operating system and then use it without manual intervention.
- DSA disks and the DUDRIVER do not accept physical QIO data transfers or seek operations.

3.2.10.1 Bad Block Replacement and Forced Errors for DSA Disks

Disks that are built according to the DSA specifications appear to be error free. Some number of logical blocks are always capable of recording data. When a disk is formatted, every user-addressable logical block is mapped to a functioning portion of the actual disk surface, which is known as a physical block. The physical block has the true data storage capacity represented by the logical block.

Additional physical blocks are set aside to replace blocks that fail during normal disk operations. These extra physical blocks are called **replacement blocks**. Whenever a physical block to which a logical block is mapped begins to fail, the associated logical block is remapped (revectored) to one of the replacement blocks. The process that revectors logical blocks is called a **bad block replacement** operation. Bad block replacement operations use data stored in a special area of the disk called the **Replacement and Caching Table (RCT)**.

When a drive-dependent error threshold is reached, the need for a bad block replacement operation is declared. Depending on the controller involved, the bad block replacement operation is performed either by the controller itself (as is the case with HSCs) or by the host (as is the case with UDAs). In either case, the same steps are performed. After inspecting and altering the RCT, the failing block is read and its contents are stored in a reserved section of the RCT.

The design goal of DSA disks is that this read operation proceeds without error and that the RCT copy of the data is correct (as it was originally written). The failing block is then tested with one or more data patterns. If no errors are encountered in this test, the original data is copied back to the original block and no further action is taken. If the data-pattern test fails, the logical block is revectored to a replacement block. After the block is revectored, the original data is copied back to the revectored logical block. In all these cases, the original data is preserved and the bad block replacement operation occurs without the user being aware that it happened.

However, if the original data cannot be read from the failing block, a best attempt copy of the data is stored in the RCT and the bad block replacement operation proceeds. When the time comes to write-back the original data, the best attempt data (stored in the RCT) is written back with the **forced error** flag set. The forced error flag is a signal that the data read is questionable. Reading a block that contains a forced error flag causes the status SS$_FORCEDERROR to be returned. This status is displayed by the following message:

```
%SYSTEM-F-FORCEDERROR, forced error flagged in last sector read
```

Writing into a block always clears the forced error flag.

Note that most VMS utilities and DCL commands treat the forced error flag as a fatal error and terminate operation when they encounter it. However, the Backup Utility (BACKUP) continues to operate in the presence of most errors, including the forced error. BACKUP continues to process the file, and the forced error flag is lost. Thus, data that was formerly marked as questionable may become "correct" data.

System managers (and other users of BACKUP) should assume that forced errors reported by BACKUP signal possible degradation of the data.

To determine what, if any, blocks on a given disk volume have the forced error flag set, use the ANALYZE /DISK_STRUCTURE /READ_CHECK command, which invokes the Verify Utility. The Verify Utility reads every logical block allocated to every file on the disk and then reports (but ignores) any forced error blocks encountered.

3.2.11 VAXstation 2000 and MicroVAX 2000 Disk Driver

The VAXstation 2000 and MicroVAX 2000 disk driver supports some DSA disk operation. In particular, the driver supports block revectoring and bad block replacement. This provides the system with a logically perfect disk medium.

Like other DSA disks, if a serious error occurs during a replacement operation, the disk is write-locked to prevent further changes. This is done to preserve data integrity and minimize damage that could be caused by failing hardware. Unlike other DSA disks, there is no visible indication on the drive itself that the disk is write-locked. However, the following indicators help you determine that the disk has become write-protected:

- ERRFMT messages show that the disk is write-locked.
- The disk enters mount verification and hangs.
- DCL command SHOW DEVICE output shows that the disk is write-locked.
- Error messages from programs and utilities attempting to write to the disk.

If the disk becomes write-locked, you should use the following procedure:

1. Shut down the system.
2. Use standalone BACKUP to create a full backup of the disk.
3. Format the disk with the disk formatter.
4. Restore the disk from the backup using standalone BACKUP. Note that any files with sectors flagged with a forced error may be corrupted and may need to be restored from a previous backup.

If errors occurring during replacement operations persist, call Digital Customer Services.

3.2.12 SCSI Disk Class Driver

The VAXstation 3100, 3520, and 3540 contain a SCSI bus that provides access to as many as seven SCSI disks. The SCSI disk class driver controls SCSI disks on all of the above systems. Although, SCSI disks do not conform to DSA, they do support the following error recovery features:

- Static and dynamic bad block replacement (BBR)
- Error correcting code (ECC)
- Reexecution of read or write operations within the SCSI drive
- Reexecution of read or write operations by the SCSI disk class driver

All SCSI disks supplied by Digital implement the REASSIGN BLOCKS command which relocates data for a specific logical block to a different physical location on the disk. The SCSI disk class driver reassigns the block in the following instances: (1) when the retry threshold is exceeded during an attempt to read or write a block of data on the disk or (2) when an irrecoverable error occurs during a write operation.

Unlike DSA, there is no forced error flag in SCSI. Blocks that produce irrecoverable errors during read operations are not reassigned in order to prevent undetected loss of user data. Instead, the SCSI disk class driver returns the SS$_PARITY status whenever a read operation results in an irrecoverable error.

3.3 Disk Driver Device Information

You can obtain information on all disk device characteristics by using the Get Device/Volume Information ($GETDVI) system service (see the *VMS System Services Reference Manual*).

$GETDVI returns disk characteristics when you specify the item codes DVI$_DEVCHAR and DVI$_DEVCHAR2. Table 3–2 lists the possible characteristics for disk devices.

Table 3–2 Disk Device Characteristics

Characteristic[1]	Meaning
Dynamic Bits (Conditionally Set)	
DEV$M_AVL	Device is online and available.
DEV$M_CDP[2,3]	Dual-path device with two UCBs.
DEV$M_CLU[2]	Device is available clusterwide.
DEV$M_2P[2]	Device is dual-pathed.
DEV$M_FOR	Device is foreign.

[1]Defined by the $DEVDEF macro.

[2]These bits are located in DVI$_DEVCHAR2.

[3]MASSBUS only.

(continued on next page)

Table 3–2 (Cont.) Disk Device Characteristics

Characteristic[1]	Meaning
Dynamic Bits (Conditionally Set)	
DEV$M_MNT	Volume is mounted.
DEV$M_RCK	Perform data check all reads.
DEV$M_WCK	Perform data check all writes.
DEV$M_MSCP[2]	Device is accessed using the mass storage control protocol.
DEV$M_RCT	Disk contains Replacement and Caching Table.
DEV$M_SRV[2]	For a VAXcluster, device is served by the MSCP server.
Static Bits (Always Set)	
DEV$M_FOD	Device is file-oriented.
DEV$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.

[2]These bits are located in DVI$_DEVCHAR2.

DVI$_DEVBUFSIZ returns the buffer size. The buffer size is the default to be used for disk transfers (this default is normally 512 bytes). DVI$_DEVTYPE and DVI$_DEVCLASS return the device type and class names, which are defined by the $DCDEF macro. The disk model determines the device type. For example, the device type for the RA81 is DT$_RA81. (Foreign device types take the form DT$_FD1 through DT$_FD8.) The device class for disks is DC$_DISK.

DVI$_CYLINDERS returns the number of cylinders per volume (that is, per disk), DVI$_TRACKS returns the number of tracks per cylinder, and DVI$_SECTORS returns the number of sectors per track. Values are returned as four-byte decimal numbers.

DVI$_MAXBLOCK returns the maximum number of blocks (1 block = 512 bytes) that can be contained on the volume (that is, on the disk). Values are returned as four-byte decimal numbers. This information can be used, for example, to determine the density of an RX02 diskette (single density = 494 blocks, double density = 988 blocks).

3.4 Disk Function Codes

VMS disk drivers can perform logical, virtual, and physical I/O functions. Foreign-mounted devices do not require privilege to perform logical and virtual I/O requests.

Logical and physical I/O functions allow access to volume storage and require only that the issuing process have access to the volume. However, DSA disks and the disk class driver (DUDRIVER) do not accept physical QIO data transfers or seek operations.

Note: The results of logical and physical I/O operations are unpredictable if an ancillary control process (ACP) or extended QIO processing (XQP) is present.

Virtual I/O functions require an ACP for Files–11 On-Disk Structure Level 1 files or an XQP for Files–11 On-Disk Structure Level 2 files. Virtual I/O functions must be executed in a prescribed order. First, you create and access a file, then you write information to that file, and lastly you deaccess the file. Subsequently, when you access the file, you read the information, and then deaccess the file. Delete the file when the information is no longer useful.

Non-DSA disk devices can read or write up to 65,535 bytes in a single request. DSA devices connected to an HSC50 can transfer up to 4 billion bytes in a single request. In all cases, the maximum size of the transfer is limited by the number of pages that can be faulted into the process's working set, and then locked into physical memory. (The disk driver is responsible for any memory management functions of this type.) The size of the transfer does not affect the applicable quotas (direct I/O count, buffered I/O count, and AST count limit). These quotas refer to the number of outstanding I/O operations of each type, not the size of the I/O operation being performed.

The volume to which a logical or virtual function is directed must be mounted for the function actually to 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 1 describes the QIO level interface to the disk device ACP.

Table 3–3 Disk I/O Functions

Function Code	Arguments	Type[1]	Function Modifiers	Function
IO$_ACCESS	P1, [P2],[P3],[P4],[P5]	V	IO$M_CREATE IO$M_ACCESS	Search a directory for a specified file and access the file if found.
IO$_ACPCONTROL	P1,[P2],[P3],[P4],[P5]	V	IO$M_DMOUNT	Perform miscellaneous control functions.
IO$_AVAILABLE		P		Clear volume valid; make DSA units available.
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$_DEACCESS	P1,[P2],[P3],[P4],[P5]	V		Deaccess a file and, if specified, write final attributes in the file header.
IO$_DELETE	P1,[P2],[P3],[P4],[P5]	V	IO$M_DELETE	Remove a directory entry or file header, or both.
IO$_FORMAT	P1	P		Set density (RX02 only).
IO$_MODIFY	P1,[P2], [P3],[P4],[P5]	V		Modify the file attributes or allocation, or both.
IO$_PACKACK		P		Update UCB fields if RX02; initialize volume valid on other devices. Bring DSA units online.
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$M_INHSEEK[3]	Read physical block.[5]
IO$_READVBLK	P1,P2,P3	V	IO$M_DATACHECK[2] IO$M_INHRETRY	Read virtual block.
IO$_SEARCH	P1	P		Search for specified block or sector (only for TU58).
IO$_SEEK	P1	P		Seek to specified cylinder.[5]
IO$_SENSECHAR		P		Sense the device-dependent characteristics and return them in the I/O status block.
IO$_SENSEMODE		L		Sense the device-dependent characteristics and return them in the I/O status block.

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

[2] Not for RX01 and RX02

[3] Not for TU58, RX01, RX02, RB02, and RL02

[5] Not for DSA and SCSI disks

(continued on next page)

Table 3–3 (Cont.) Disk I/O Functions

Function Code	Arguments	Type[1]	Function Modifiers	Function
IO$_SETPRFPTH	P1	P	IO$M_FORCEPTH	Specifies a preferred path for DSA disks.
IO$_UNLOAD		P		Clear volume valid; make DSA units available and spin down the volume.
IO$_WRITECHECK	P1,P2,P3	P		Verify data written to disk by a previous write QIO.[2]
IO$_WRITELBLK	P1,P2,P3	L	IO$M_DATACHECK[2] IO$M_ERASE IO$M_INHRETRY	Write logical block.
IO$_WRITEPBLK	P1,P2,P3	P	IO$M_DATACHECK[2] IO$M_ERASE IO$M_INHRETRY IO$M_INHSEEK[3] IO$M_DELDATA[4]	Write physical block.[5]
IO$_WRITEVBLK	P1,P2,P3	V	IO$M_DATACHECK[2] IO$M_ERASE IO$M_INHRETRY	Write virtual block.

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

[2]Not for RX01 and RX02

[3]Not for TU58, RX01, RX02, RB02, and RL02

[4]RX02 only

[5]Not for DSA and SCSI disks

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

- 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 1 for more information on these functions.

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

- P1—The starting virtual address of the buffer that is to receive the data from 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, RX211, or UDA50.
- P3—The starting virtual/logical disk address of the data to be transferred in a read operation; or, in a write operation, the disk address of the area that is to receive the data.

  In a virtual read or write operation, 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 using mapping windows that are set up by the file system ACP process.)

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

The function-dependent arguments for IO$_WRITEVBLK, IO$_WRITELBLK, and IO$_WRITEPBLK functions that include the IO$M_ERASE function modifier are as follows:

- P1—The starting virtual address of the buffer that contains a four-byte, user-specified erase pattern. If the P1 address is 0, a longword of 0 will be used for the erase pattern. If the P1 address is nonzero, the contents of the four bytes starting at that address will be used as the erase pattern. Digital recommends that the user specify a P1 address of 0 to lower system overhead.

Note: DSA disk controllers provide controlled, assisted erasing for the IO$M_ERASE modifier (with virtual and logical write functions) only when the erase pattern is all 0s. If a nonzero erase pattern is used, there is a significant performance degradation with these disks. DSA disks do not accept physical QIO transfers.

- P2—The number of bytes of erase pattern to write to the disk. The number specified is rounded up to the next highest block boundary (512 bytes).
- P3—The starting virtual, logical, or physical disk address of the data to be erased.

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

- P1—The starting virtual address of the buffer that is to receive the data in a read operation; or, in a write operation, the starting virtual address of the buffer that is to be written on the disk. Passed by reference.
- P2—The number of bytes that are to be read from the disk, or written from memory to the disk. Passed by value. An even number must be specified if the controller is an RK611, RL11, or UDA50.

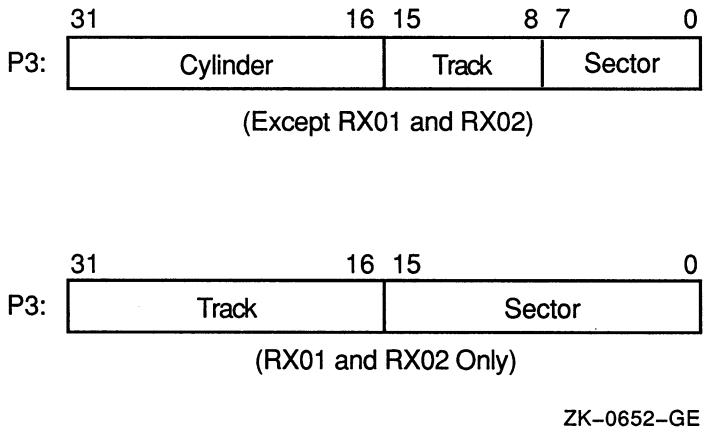
- P3—The starting physical disk address of the data to be read in a read operation; or, in a write operation, the starting physical address of the disk area that is to receive the data. Passed by value. The address is expressed as sector, track, and cylinder in the format shown in Figure 3–3. (On the RX01 and RX02, the high word specifies the track number rather than the cylinder number.) Check the UCB of a currently mounted device to determine the maximum physical address value for that type of device.

Note: On the RB80 and RM80, do not address cylinders 560 and 561. These two cylinders are used for diagnostic testing only.

The function-dependent argument for IO$_SEARCH is as follows:

- P1—The physical disk address where the tape is positioned. The address is expressed as sector, track, and cylinder in the format shown in Figure 3–3.

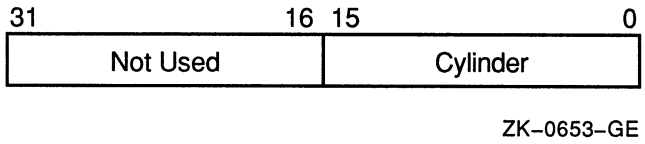
Figure 3–3 Starting Physical Address



The function-dependent argument for IO$_SEEK is as follows:

- P1—The physical cylinder number where the disk heads are positioned. The address is expressed in the format shown in Figure 3–4.

Figure 3–4 Physical Cylinder Number Format



The function dependent argument for IO$_FORMAT is as follows:

- P1—The density at which an RX02 diskette is reformatted (see Section 3.4.4).

3.4.1 Read

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

The VMS operating system provides the following 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, that 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 same way as for a file-structured device.

Three function-dependent arguments are used with these codes: P1, P2, and P3. These arguments are described in 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 completes. A data check operation is also performed if the volume that has been 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 and RX02 drivers do not support the data check function.

If IO$M_DATACHECK is specified with a read function code to a TU58, or if the volume read has the characteristic "data check all reads," a read check operation is performed. This alters certain TU58 hardware parameters when the tape is read. (The read threshold in the data recovery circuit is increased; if the tape has any weak spots, errors are detected.)

The data check function modifier to a disk or tape can return five error codes in the I/O status block:

SS$_CTRLERR SS$_DRVERR SS$_MEDOFL
SS$_NONEXDRV SS$_NORMAL

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

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

The VMS operating system provides the following 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 same way as for a file-structured device.

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

The data check function modifier (IO$M_DATACHECK) can be used with all write operations. If this modifier is specified, a data check operation is performed after the write operation completes. 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 and RX02 drivers do not support the data check function.

If IO$M_DATACHECK is specified with a write function code to a TU58, or if the volume written has the characteristic "data check all writes," a write check operation is performed. The write check verifies data written on the tape. First, the specified data is written on the tape. Then the tape is reversed and the TU58 controller reads the data internally to perform a checksum verification. If the checksum verification is unsuccessful after eight attempts, the write check operation is aborted and an error status is returned.

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

The write deleted data function modifier (IO$M_DELDATA) can be used with the write physical block (IO$_WRITEPBLK) function to the RX02. If this modifier is specified, a deleted data address mark instead of the standard data address mark is written preceding the data. Otherwise, the operation of the IO$_WRITEPBLK function is the same; write data is transferred to the disk. When a successful read operation is performed on this data, the status code SS$_RDDELDATA is returned in the I/O status block rather than the usual SS$_NORMAL status code.

The IO$M_ERASE function modifier can be used with all write function codes to erase a user-selected part of a disk. This modifier propagates an erase pattern through the specified range. Section 3.4 describes the write function arguments to be used with IO$M_ERASE.

3.4.3 Sense Mode

Sense mode operations obtain current disk device-dependent characteristics that are returned to the caller in the second longword of the I/O status block (see Figure 3–6). The VMS operating system provides the following function codes:

- IO$_SENSEMODE—Sense characteristics
- IO$_SENSECHAR—Sense characteristics

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

3.4.4 Set Density

The set density function assigns a new density to an entire RX02 floppy diskette. The diskette is also reformatted: new data address marks are written (single or double density) and all data fields are zeroed. Set density is a physical I/O function and requires the access privilege necessary to perform physical I/O. The following function code is provided:

- IO$_FORMAT

IO$_FORMAT takes the following function-dependent argument:

- P1—The density at which the diskette is reformatted:

  0 = single density (default)
  1 = single density
  2 = double density

The set density operation should not be interrupted before it is completed (about 15 seconds). If the operation is interrupted, the resulting diskette might contain illegal data address marks in both densities. The diskette must then be completely reformatted and the function reissued.

3.4.5 Search

The search function positions a TU58 magnetic tape to the block specified. Search is a physical I/O function and requires the access privilege necessary to perform physical I/O. The VMS operating system provides a single function code:

- IO$_SEARCH

This function code takes the following function-dependent argument:

- P1—Specifies the block where the read/write head will be positioned. The low byte contains the sector number in the range 0 to 127; the high byte contains the track number in the range 0 to 3.

IO$_SEARCH can save time between read and write operations. For example, nearly 30 seconds are required to completely rewind a tape. If the last read or write operation is near the end of the tape and the next operation is near the beginning of the tape, the search operation can begin after the last operation completes, and the tape will rewind while the process is otherwise occupied. (The search QIO is not completed until the search is completed. Consequently, if a $QIOW system service request is issued, the process will be held up until the search is completed.)

3.4.6 Pack Acknowledge

The pack acknowledge 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. If directed to an RX02 drive, pack acknowledge also determines the diskette density and updates the device-dependent information returned by $GETDVI item codes DVI$_CYLINDERS, DVI$_TRACKS, DVI$_SECTORS, DVI$_DEVTYPE, DVI$_CLASS, and DVI$_MAXBLOCK. If directed to a DSA disk, pack acknowledge also sends the online packet to the controller. The following 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 DCL commands INITIALIZE or MOUNT are issued.

For DSA disks, the IO$_PACKACK function locks the drive's port selector on the port that initiated the pack acknowledge function.

In addition, the IO$_PACKACK function updates device-dependent information about DSA disks returned by $GETDVI.

3.4.7 Unload

The unload function clears the volume valid bit for all disk drives, makes DSA disks available, and issues an unload command to the drive (spins down the volume). The unload function reverses the function performed by pack acknowledge (see Section 3.4.6). The following function code is provided:

- IO$_UNLOAD

This function takes no function-dependent arguments.

3.4.8 Available

The available function clears the volume valid bit for all disk drives; that is, it reverses the function performed by pack acknowledge (see Section 3.4.6). No unload function is issued to the drive. Therefore, those drives capable of spinning down do not spin down. The following function code is provided:

- IO$_AVAILABLE

This function takes no function-dependent arguments.

3.4.9 Seek

The seek function directs the read/write heads to move to the cylinder specified in the P1 argument (see Sections 3.2.7 and 3.4, and Figure 3–4).

3.4.10 Write Check

The write check function verifies that data was written to disk correctly. The data to be checked is addressed using physical disk addressing (sector, track, and cylinder) (see Figure 3–3). If the request is directed to a DSA disk, you must specify a logical block number, even though IO$_WRITECHECK is a physical I/O function. The following function code is provided:

- IO$_WRITECHECK

A write QIO must be used to write data to disk before you enter this command. IO$_WRITECHECK then reads the same block of data and compares it with the data in the specified buffer. Three function-dependent arguments are used with this code: P1, P2, and P3. These arguments are described in Section 3.4.

IO$_WRITECHECK is similar to the IO$M_DATACHECK function modifier for write QIOs, except that IO$_WRITECHECK does not write the data to disk; it is specified after data is written by a separate write QIO. Nonprivileged processes can use the IO$M_DATACHECK modifier with IO$_WRITEVBLK (which does not require access privilege) to determine whether data is written correctly. The RX01 and RX02 drivers do not support the write check function.

The write check function and the data check function modifier to a TU58 can return six error codes in the I/O status block: SS$_NORMAL, SS$_CTRLERR, SS$_DRVERR, SS$_MEDOFL, SS$_NONEXDRV, and SS$_WRTLCK.

3.4.11 Set Preferred Path

The set preferred path function specifies a preferred path for DSA disks. This includes RA-series disks and disks accessed through the MSCP server. If a preferred path is specified for a disk, the MSCP disk class drivers (DUDRIVER and DSDRIVER) use the path as their first attempt to locate the disk and bring it on line as a result of a DCL command MOUNT or failover of an already mounted disk. In addition, you can initiate failover of a mounted disk in order to force the disk to the preferred path, or to use load-balancing information for disks accessed through MSCP servers.

The function code is:

IO$_SETPRFPTH

The following is the function modifier:

- IO$M_FORCEPATH—causes the disk class driver to select the server path with the highest load available rating.

The P1 parameter contains the address of a counted ASCII string (.ASCIC). This string is the node name of the HSC or VMS system that is the preferred path. The node name must match an existing node that is known to the local node and if the node is a VMS system, it must be running the MSCP server. This function does not move the disk to the preferred path.

The PHYS_IO privilege is required for IO$_SETPRFPTH and IO$M_FORCEPATH.

The following example shows the use of IO$_SETPRFPTH:

```
        $assigndef
        $qiodef
        $iodef
        $exitdef

dev:    .ascid  /$254$DUA48:/

chnl:   .word   0

node:   .ascic  /HSC001/

        .entry  start,0

        $assign_s       devnam=dev,-
                        chan=chnl
        blbc    r0,done

        $qiow_s         chan=chnl,-
                        func=#IO$_SETPRFPTH,-
                        p1=node

done:
        $exit_s r0

        .end  start
```

This updates the local node I/O database to indicate that node HSC001 is the preferred path for DUA48.

3.4.11.1 Forcing a Path Change

You can move a disk that is already mounted to its preferred path by specifying the IO$M_FORCEPATH modifier. If a preferred path has not been specified for a disk that is accessed through the MSCP server, the IO$M_FORCEPATH function causes the disk class driver to use load-balancing information to select the server path with the highest load available rating.

IO$M_FORCEPATH does not accept any arguments. If you intend to move a disk to its preferred path, you must specify the preferred path in a separate $QIO function.

The following example shows use of the IO$M_FORCEPATH function modifier:

```
        $assigndef
        $qiodef
        $iodef
        $exitdef

dev:    .ascid  /$254$DUA197:/

chnl:   .word   0

        .entry  start,0

        $assign_s       devnam=dev,-
                        chan=chnl
        blbc    r0,done

        $qiow_s         chan=chnl,-
                        func=#<IO$_SETPRFPTH!IO$M_FORCEPATH>

done:
        $exit_s r0

        .end    start
```

Note that forcing a path change places the disk in mount verification. New I/O requests are suspended until mount verification is complete.

3.4.11.2 Using IO$_SETPRFPTH with Disks Dual Pathed Between HSCs

You can use the IO$_SETPRFPTH and IO$M_FORCEPATH functions to load balance disks that are dual pathed between HSCs. The IO$M_FORCEPATH function initiates failover of the disk on all nodes that have it mounted and that have a direct path to the HSCs. Since the node that issues the IO$M_FORCEPATH might not be the first one to attempt failover of the disk, it is essential that all nodes that have direct connections to the HSCs specify the same preferred path for the disk. Only one node should issue the IO$M_FORCEPATH request.

3.4.11.3 Using IO$_SETPRFPTH with Disks Dual Pathed Between VMS Systems

You can use IO$M_FORCEPATH to load balance RA-series disks that are dual pathed between VMS systems running the MSCP server. Both serving nodes should specify the same preferred path. In order to move the disk between VMS systems, the system that currently has the disk on line through its local controller should issue the IO$M_FORCEPATH request. The disk must be mounted on both serving nodes.

3.4.11.4 Using IO$_SETPRFPTH with Disks Accessed Through MSCP Servers

You can specify a preferred path for disks that are accessed through MSCP servers. However, this specification overrides any load-balancing decisions.

Note that if a disk can be accessed through both HSC and MSCP servers, you need not specify the HSC as a preferred path. HSC paths are always preferred to server paths.

Using IO$M_FORCEPATH without a preferred path causes the disk class driver to move the disk to the server with the highest available capacity.

3.4.11.5 Using IO$_SETPRFPTH with Phase I Volume Shadowing

You can specify IO$_SETPRFPTH for shadow set members, but not for virtual units. IO$M_FORCEPATH is not supported for shadow set members or virtual units.

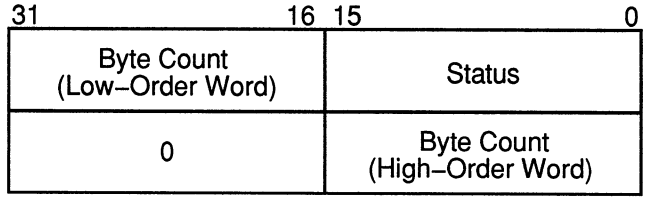
3.4.11.6 Using IO$_SETPRFPTH with Phase II Volume Shadowing

IO$_SETPRFPTH and IO$M_FORCEPATH are supported for shadow set members but not for virtual units.

3.5 I/O Status Block

Figure 3–5 shows the I/O status block (IOSB) for all disk device QIO functions except Sense Mode. Figure 3–6 shows the I/O status block for the Sense Mode function. Appendix A lists the status messages for all functions and devices. (The *VMS System Messages and Recovery Procedures Reference Manual* provides explanations and suggested user actions for these messages.)

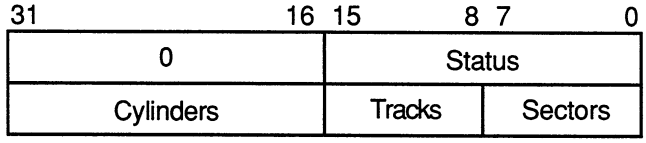
Figure 3–5 IOSB Contents



ZK–0656–GE

The byte count is a 32-bit integer that gives the actual number of bytes transferred to or from the process buffer.

Figure 3–6 IOSB Contents for the Sense Mode Function



ZK–0657–GE

The second longword of the I/O status block for the Sense Mode function returns information about the cylinder, track, and sector configurations for the particular device.

3.6 Disk Driver Programming Example

This sample program (Example 3–1) provides an example of optimizing access time to a disk file. The program creates a file using VMS 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 ($QIO) system service.

Example 3–1 Disk Program Example

```
; ****************************************************************
;
        .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:                              ;
        $FAB    ALQ = 100,-             ;Initial file size is to be
                -                       ;100 blocks
                FAC = PUT,-             ;File Access Type is output
                FNA = FILE_NAME,-       ;File name string address
                FNS = FILE_SIZE,-       ;File name string size
                FOP = CTG,-             ;File is to be contiguous
                MRS = 512,-             ;Maximum record size is 512
                -                       ;bytes
                NAM = NAM_BLOCK,-       ;File name block address
                ORG = SEQ,-             ;File organization is to be
                -                       ;sequential
                REM = FIX               ;Record format is fixed length
;
; Allocate file information block.
;
;       A file information block is required as an argument in the
;       Queue I/O system service call that accesses a file.
;
FIB_BLOCK:                              ;
        .BLKB   FIB$K_LENGTH            ;

;
; Allocate file information block descriptor.
;
FIB_DESCR:                              ;
        .LONG   FIB$K_LENGTH            ;Length of the file
                                        ;information block
        .LONG   FIB_BLOCK               ;Address of the file
                                        ;information block
```

(continued on next page)

Example 3–1 (Cont.) Disk Program Example

```
;
; Allocate File Name Block
;
;       A file name block is required by RMS-32 to return information
;       concerning a file (for example, the resultant file name 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 address 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.
;
```

(continued on next page)

Example 3–1 (Cont.) Disk Program Example

```
IO_STATUS:                              ;
        .BLKQ   1                       ;
;
; Allocate output record buffer.
;
RECORD_BUFFER:                          ;
        .BLKB   512                     ;Record buffer is 512 bytes
;
; ****************************************************************
;
;                       Start Program
;
; ****************************************************************

;
; The purpose of the program is to create a file called MYDATAFIL.DAT
; using RMS-32; store information concerning the file; write 100
; records, each containing its record number in every byte;
; close the file; and then access, 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 = 0, 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 = 0, creation
                                        ;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

;
; Now use RMS-32 to write the record into the newly created file.
;
```

(continued on next page)

Example 3–1 (Cont.) Disk Program Example

```
        $PUT    RAB = RAB_BLOCK         ;Put record in file
        BLBC    R0,30$                  ;If low bit = 0, 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:                                 ;
        $ASSIGN_S DEVNAM = DEVICE_DESCR,-  ;Assign a channel to file
                CHAN = DEVICE_CHANNEL   ;device
        BLBC    R0,20$                  ;If low bit = 0, assign
                                        ;failure
        MOVL    #FIB$M_NOWRITE!FIB$M_WRITE,- ;Set for read/write
                FIB_BLOCK+FIB$L_ACCTL   ;access
        $QIOW_S CHAN = DEVICE_CHANNEL,- ;Access file on device channel
                FUNC = #IO$_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 = 0, access
                                        ;failure
        MOVZWL  IO_STATUS,R0            ;Get final I/O completion
                                        ;status
```

(continued on next page)

Example 3–1 (Cont.) Disk Program Example

```
        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 two 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 having the old contents of record 1, and so forth. After
; the example has been run, it is suggested that the file dump
; utility be used to verify the change in data positioning.
;
30$     MOVZBL  #1,R6                   ;Set starting record (block)
                                        ;number
;
; Read next two records into block buffer.
;
40$:    $QIO_S  CHAN = DEVICE_CHANNEL,- ;Read next two records from
                -                       ;file channel
                FUNC = #IO$_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.
;
```

(continued on next page)

Example 3–1 (Cont.) Disk Program Example

```
        $QIOW_S CHAN = DEVICE_CHANNEL,- ;Write even-numbered record in
                -                       ;odd slot
                FUNC = #IO$_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 = 0, service
                                        ;failure
        MOVZWL  IO_STATUS,R0            ;Get final I/O completion
                                        ;status
        BLBC    R0,70$                  ;If low bit = 0, I/O function
        RSB                             ;failure
;
; 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
```

4 Laboratory Peripheral Accelerator Driver

This chapter describes the VMS laboratory peripheral accelerator (LPA11-K) driver and the high-level language procedure library that interfaces with it. The procedure library is implemented with callable assembly language routines that translate arguments into the format required by the LPA11-K driver and that handle buffer chaining operations. Routines for loading the microcode and initializing the device are also described.

Refer to the *LPA11-K Laboratory Peripheral Accelerator User's Guide* for additional information.

4.1 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 processor through the UNIBUS adapter.

The LPA11-K is a fast, flexible microprocessor subsystem designed for applications requiring high-speed, concurrent data acquisition and data reduction. The LPA11-K allows aggregate analog input and output rates of up to 150,000 samples per second. The maximum aggregate digital input and output rate is 15,000 samples per second.

Table 4–1 lists the useful minimum and maximum LPA11-K configurations supported by the VMS operating system.

4.1.1 LPA11-K Modes of Operation

The LPA11-K operates in two modes: dedicated and multirequest.

In dedicated mode, only one user (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 4–1 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
1 of the following:	2 AD11-K A/D converters
AD11-K A/D converter	2 AM11-K multiplexers for AD11-K converters
AA11-K A/D converter	1 AA11-K D/A converter
DR11-K digital I/O register	5 DR11-K digital I/O registers

In multirequest mode, sampling from all of the devices listed in Table 4–1 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.

4.1.2 Errors

The LPA11-K returns the following 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 that affect all LPA11-K activity (SS$_CTRLERR)

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.

The LADRIVER times out all $QIOs after two seconds if they have not completed. The driver does not provide any parameters that allow the user to change the length of the timeout.

The timeout period applied to all $QIOs can be changed with the following PATCH commands executed from a privileged account:

```
$ PATCH SYS$SYSTEM:LADRIVER.EXE/OUTPUT=SYS$SYSTEM:LADRIVER.EXE
PATCH> SET ECO 25
PATCH> REPLACE/INSTRUCTION LA$TIMEOUT_VALUE
OLD> 'PUSHL     I^#00000002'
OLD> EXIT
NEW> 'PUSHL     I^#0000003C'
NEW> EXIT
PATCH> UPDATE
PATCH> EXIT
```

Substitute the desired timeout value for the "0000003C" in the example above. When you reboot, the system loads the new copy of the driver containing the new timeout value.

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.

4.2 Supporting Software

The LPA11-K is supported by a device driver, a high-level language procedure library of support routines, and routines for loading the microcode and initializing the device. The system software and support routines provide a control path for synchronizing the use of buffers, specifying requests, and starting and stopping requests; the actual data algorithms for the laboratory data acquisition I/O devices are accomplished by the LPA11-K.

The LPA11-K driver and the associated I/O interface have the following features:

- They permit multiple LPA11-K subsystems on a single UNIBUS adapter.
- They operate as an integral part of the VMS operating system.
- They can be loaded on a running VMS operating system without relinking the executive.
- They handle I/O requests, function dispatching, UNIBUS adapter 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.

- Because the LPA11-K chains buffer areas automatically, a start data transfer request can transfer an infinite and noninterrupted 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:

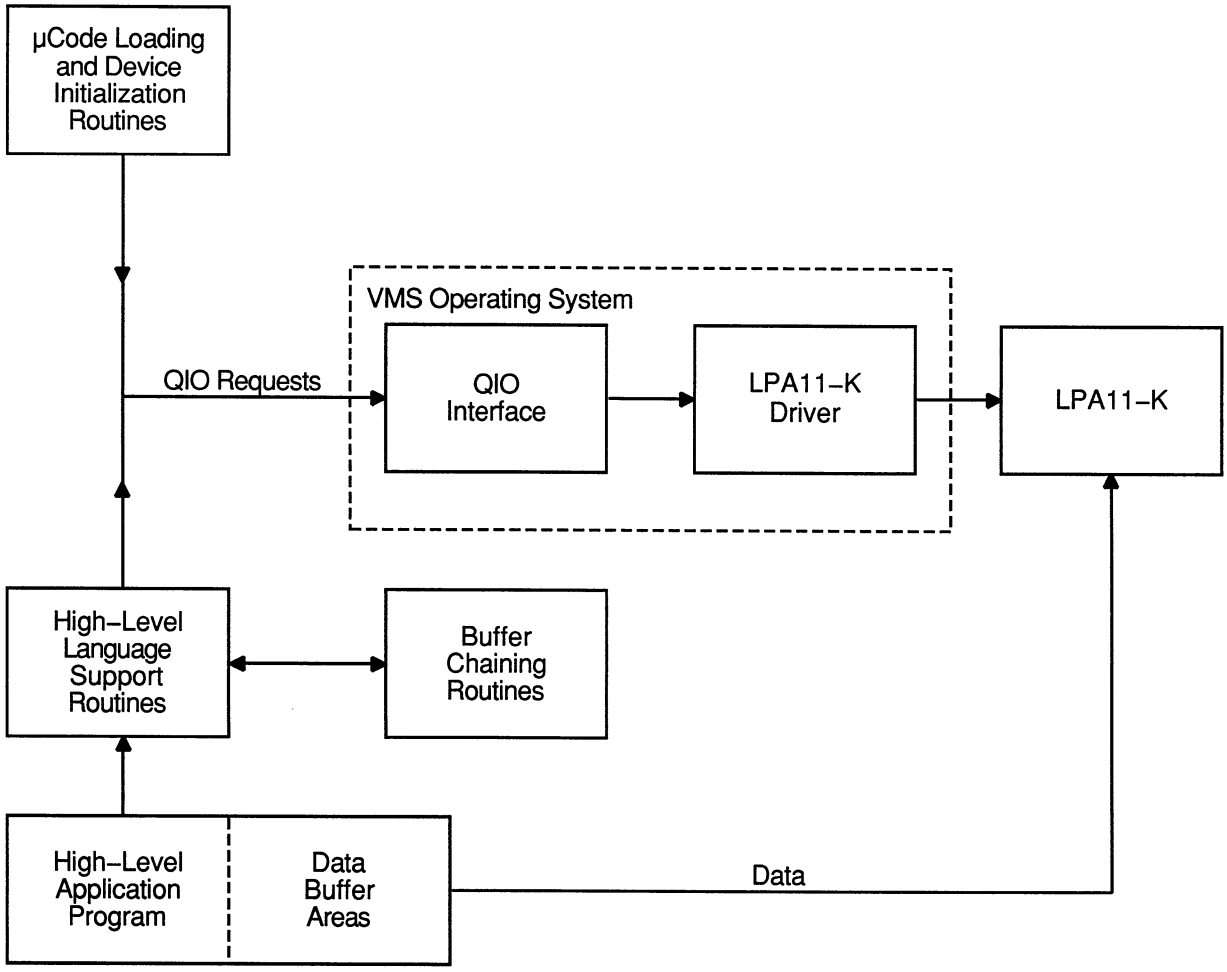
- They translate arguments provided in the high-level language calls into the format required for the Queue I/O interface.
- They provide a buffer chaining capability for a multibuffering environment by maintaining queues of used, in use, and available buffers.
- They adhere to all VMS conventions for calling sequences, use of shareable resources, and reentrancy.
- They can be part of a resident global library, or they can be linked into a process image as needed.

The routines for loading microcode and initializing devices have the following features:

- They execute, as separate processes, images that 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 at the request of the user or an operator.
- 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 4–1 shows the relationship of the supporting software to the LPA11-K.

Figure 4–1 Relationship of Supporting Software to LPA11-K



4.3 LPA11-K Device Information

You can obtain information on all peripheral data acquisition devices on the LPA11-K internal I/O bus by using the Get Volume Information ($GETDVI) system service. (See the *VMS System Services Reference Manual.*)

$GETDVI returns device characteristics when you specify the item codes DVI$_DEVCHAR and DVI$_DEVDEPEND. Tables 4–2 and 4–3 list these characteristics. The $DEVDEF macro defines the device-independent characteristics; the $LADEF macro defines the device-dependent characteristics. Device-dependent 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 4–3.

DVI$_DEVCLASS and DVI$_DEVTYPE return the device class and device type names, which are defined by the $DCDEF macro. The device class for the LPA11-K is DC$_REALTIME; the device type is DT$_LPA11. DVI$_DEVBUFSIZ is not applicable to the LPA11-K.

Table 4–2 LPA11-K Device-Independent Characteristics

Characteristic[1]	Meaning
	Dynamic Bit (Conditionally Set)
DEV$M_AVL	Device is online and available.
	Static Bits (Always Set)
DEV$M_IDV	Device is capable of input.
DEV$M_ODV	Device is capable of output.
DEV$M_RTM	Device is real-time.
DEV$M_SHR	Device is shareable.

[1]Defined by the $DEVDEF macro.

Table 4–3 LPA11-K Device-Dependent Characteristics

Field[1]	Meaning
LA$M_MCVALID LA$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:

Value	Meaning
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.

[1]Defined by the $LADEF macro.

(continued on next page)

Table 4–3 (Cont.) LPA11-K Device-Dependent Characteristics

Field[1]	Meaning
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:

Value	Meaning
LA$V_CLOCKA LA$M_CLOCKA	Clock A
LA$V_CLOCKB LA$M_CLOCKB	Clock B
LA$V_AD1 LA$M_AD1	A/D device 1
LA$V_AD2 LA$M_AD2	A/D device 2
LA$V_DA LA$M_DA	D/A device 1
LA$V_DIO1 LA$M_DIO1	Digital I/O buffer 1
LA$V_DIO2 LA$M_DIO2	Digital I/O buffer 2
LA$V_DIO3 LA$M_DIO3	Digital I/O buffer 3
LA$V_DIO4 LA$M_DIO4	Digital I/O buffer 4
LA$V_DIO5 LA$M_DIO5	Digital I/O buffer 5

LA$V_RATE LA$S_RATE	The Clock A rate, which is set by the set clock function (IO$_SETCLOCK), is one of the following values:

Value	Meaning
0	Stopped
1	1 MHz
2	100 kHz
3	10 kHz
4	1 kHz
5	100 Hz
6	Schmidt trigger
7	Line frequency

[1] Defined by the $LADEF macro.

(continued on next page)

Table 4–3 (Cont.) LPA11-K Device-Dependent Characteristics

Field[1]	Meaning
LA$V_PRESET LA$S_PRESET	The Clock A preset value set by the set clock function (IO$_SETCLOCK). (The value is in two's complement form in the range 0 through 65,535.) The clock rate divided by the clock preset value yields the clock overflow rate.

[1]Defined by the $LADEF macro.

4.4 LPA11-K Function Codes

The LPA11-K I/O functions are as follows:

- Load the microcode into the LPA11-K.
- Start the LPA11-K microprocessor.
- Initialize the LPA11-K subsystem.
- Set the LPA11-K real-time clock rate.
- Start a data transfer request.

The first three functions are normally performed by the loader process, not by the user's data transfer program. See Section 4.5.21 for a description of the loader process interface.

The Cancel I/O on Channel ($CANCEL) system service is used to abort data transfers.

4.4.1 Load Microcode

This I/O function resets the LPA11-K and loads an image of LPA11-K microcode. Physical I/O privilege is required. The following function code is provided:

- IO$_LOADMCODE—Load microcode

The load microcode function takes the following device- or 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 (the microcode was loaded successfully) the driver sets the microcode valid bit (LA$V_MCVALID) in the device-dependent characteristics longword (see Table 4–3). If there is no match, SS$_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.

4.4.2 Start Microprocessor

This I/O function resets the LPA11-K and starts (or restarts) the LPA11-K microprocessor. Physical I/O privilege is required. The following function code is provided:

- IO$_STARTMPROC—Start microprocessor

This function code takes no device- or function-dependent arguments.

The start microprocessor function can return five error codes in the I/O status block (see Section 4.6):

SS$_CTRLERR SS$_DEVACTIVE SS$_MCNOTVALID
SS$_POWERFAIL SS$_TIMEOUT

The *LPA11-K Laboratory Peripheral Accelerator User's Guide* provides additional information on error codes.

4.4.3 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. The VMS operating system defines the following function code:

- IO$_INITIALIZE—Initialize LPA11-K

The initialize LPA11-K function takes the following device- or 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 4–3).

The initialize function can return the following 10 error codes in the I/O status block:

SS$_BUFNOTALIGN	SS$_CANCEL	SS$_CTRLERR
SS$_DEVCMDERR	SS$_INCLENGTH	SS$_INSFMAPREG
SS$_IVMODE	SS$_MCNOTVALID	SS$_POWERFAIL
SS$_TIMEOUT		

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 4.6). A program can use this characteristic to poll the LPA11-K and determine the current device configuration.

4.4.4 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.

Note: 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 affects all users.

The following function code is provided:

- IO$_SETCLOCK—Set clock

The set clock function takes the following device- or function-dependent arguments:

- P2—Mode of operation. The VMS operating system defines the following clock start mode word (hexadecimal) values:

Value	Meaning
1	KW11-K Clock A
11	KW11-K Clock B

- P3—Clock control and status. The VMS operating system defines the following clock status word (hexadecimal) values:

Value	Meaning
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 two'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 4–3).

The set clock function can return six error codes in the I/O status block (see Section 4.6):

SS$_CANCEL	SS$_CTRLERR	SS$_DEVCMDERR
SS$_MCNOTVALID	SS$_POWERFAIL	SS$_TIMEOUT

The *LPA11-K Laboratory Peripheral Accelerator User's Guide* provides additional information on error codes.

4.4.5 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. The following function code is provided:

- IO$_STARTDATA—Start data transfer request

The start data transfer request function takes the following function modifier:

- IO$M_SETEVF—Set event flag

The start data transfer request function takes the following device- or 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). This argument is used only when the buffer overrun bit (LA$M_BFROVRN) is set, that is, when a buffer overrun condition is classified as a nonfatal error.

A buffer overrun condition differs from 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 is filled or emptied, the LPA11-K obtains the index number of the next buffer to process from the user status word (USW). (See 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 4–2, when the data transfer start command is processed. After the command is accepted and data transfers begin, the driver does not access the table.

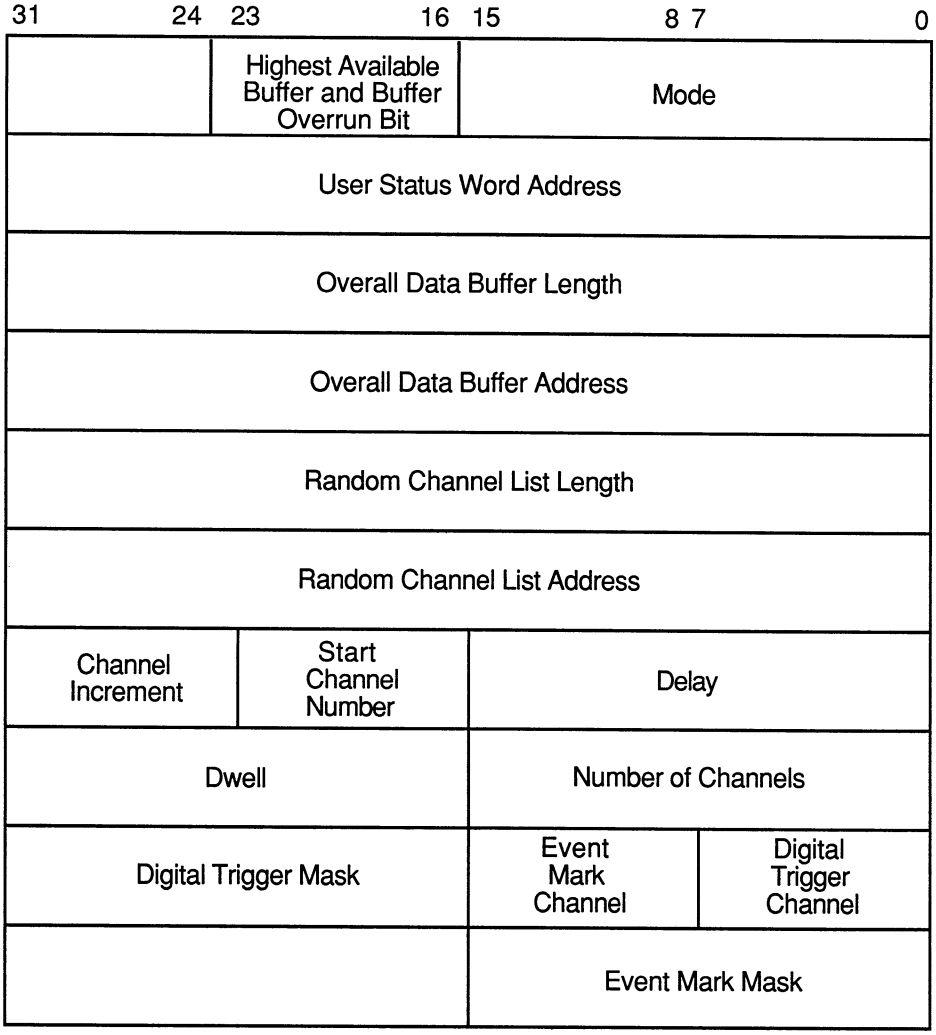
In the first longword of the data transfer command table, the first two bytes contain the LPA11-K start data transfer request mode word. (The *LPA11-K Laboratory Peripheral Accelerator User's Guide* describes the functions of this word.)

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 the *LPA11-K Laboratory Peripheral Accelerator User's Guide*). This virtual address points to a two-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.

Figure 4–2 Data Transfer Command Table



ZK–0660–GE

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 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 the *LPA11-K Laboratory Peripheral Accelerator User's Guide* for a detailed description of their functions.)

The start data transfer request function can return the following 15 error codes in the I/O status block (see Section 4.6):

SS$_ABORT	SS$_BUFNOTALIGN	SS$_CANCEL
SS$_CTRLERR	SS$_DEVCMDERR	SS$_DEVREQERR
SS$_EXQUOTA	SS$_INCLENGTH	SS$_INSFBUFDP
SS$_INSFMAPREG	SS$_INSFMEM	SS$_MCNOTVALID
SS$_PARITY	SS$_POWERFAIL	SS$_TIMEOUT

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 either by setting the event flag specified by the QIO request **efn** argument or by queuing 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$M_SETEVF) is specified, the event flag is set repeatedly: when the data transfer request is started, after each buffer completion, and when the request completes. If IO$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 the setting of event flags is not. If only event flags are used, buffer status may be lost.

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.

4.4.6 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.

To stop a data transfer, 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 the *LPA11-K Laboratory Peripheral Accelerator User's Guide*).

4.5 High-Level Language Interface

The VMS operating system supports several program-callable procedures that provide access to the LPA11-K. The formats of these calls are documented in this manual for VAX FORTRAN users. VAX MACRO users must set up a standard VMS argument block and issue the standard CALL procedure. (VAX MACRO users can also access the LPA11-K directly through the use of the device-specific QIO functions described in Section 4.4.) Users of other high-level languages must specify the proper subroutine or procedure invocation.

4.5.1 High-Level Language Support Routines

The VMS operating system provides 20 high-level language procedures for the LPA11-K. These procedures are divided into four classes. Table 4–4 lists the classes and the VAX procedures for the LPA11-K.

Table 4–4 VAX 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$XRATE	Compute clock rate and preset value
Data Buffer	LPA$IBFSTS	Return buffer status
Control	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

4.5.1.1 Buffer Queue Control

This section is provided for informational purposes only.

Buffer queue control for data transfers by LPA11-K subroutines involves the use of the following 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 indexes of all the buffers that the user has released (buffers 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 indexes 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 indexes of all buffers that are currently being processed by the LPA11-K. Normally, the IUQ contains the indexes of the following 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.)

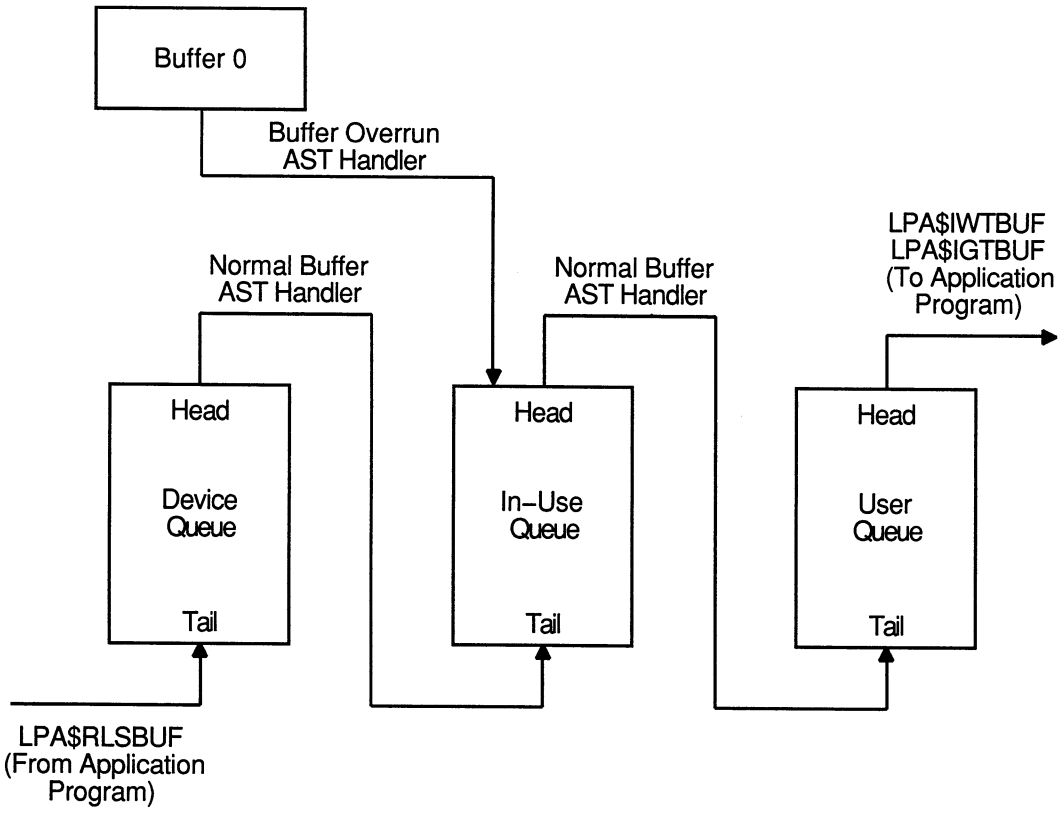
Because the LPA11-K driver requires that at least one buffer be ready when the input or output sweep is started, the user must call the LPA$RLSBUF subroutine before the sweep is initiated.

Figure 4–3 shows the flow between the buffer queues.

4.5.1.2 Subroutine Argument Usage

Table 4–5 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.

Figure 4–3 Buffer Queue Control



ZK–0661–GE

Table 4–5 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 4.6).
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 6 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 4–5 (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 can be applicable at the same time. The MODE argument length is one word.
IRATE	Specifies the clock rate as follows:

Value	Meaning
–1	Direct-coupled Schmidt trigger 1 (Clock A only)
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 two's complement of the desired number of clock ticks between clock interrupts. (The maximum value is 0, the two'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 4–5 (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 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 the LPA$IGTBUF subroutine to obtain the next buffer. If IEFN specifies an event flag, the program must call the LPA$IWTBUF subroutine 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 65,535; 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. 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.
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 VMS success or failure code of the call. The IND argument length is one longword.

4.5.2 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 subroutine call is as follows:

```
CALL LPA$ADSWP (IBUF,LBUF,[NBUF],[MODE],[DWELL],[IEFN],-
                [LDELAY],[ICHN],[NCHN],[IND])
```

Arguments are as described in Section 4.5.1.2, with the following additions:

MODE Specifies sampling options. The VMS operating system 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. This value is used when a user-supplied external sweep trigger is desired. The external trigger is supplied by the KW11-K (Schmidt trigger 1 output) to the AD11-K (external start input). If MODE=512, the user process must specify a Clock A rate of –1 for proper A/D sampling. This is nonclock-driven sampling (see Section 4.5.10). (The *LPA11-K Laboratory Peripheral Accelerator User's Guide* provides additional information on the use of external triggers.)
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 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.)
4096	Start method. If selected, the digital input start method is used. If not selected, the immediate start method is used. 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 as follows:

0 = Error in call. Possible causes are the following: LPA$SETIBF subroutine was not previously called; LPA$RLSBUF subroutine was not previously called; size of data buffers disagrees with the size computed by the LPA$SETIBF subroutine call.

1 = successful sweep started

nnn = VMS status code

4.5.3 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 subroutine call is as follows:

```
CALL LPA$DASWP (IBUF,LBUF,[NBUF],[MODE],[DWELL],[IEFN],-
     [LDELAY],[ICHN],[NCHN],[IND])
```

Arguments are as described in Section 4.5.1.2, with the following additions:

MODE Specifies the sampling options. The VMS operating system 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, the digital input start method is used. If not selected, the immediate start method is used. 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 as follows:

0 = Error in call. Possible causes are the following: LPA$SETIBF subroutine was not previously called; LPA$RLSBUF subroutine was not previously called; size of data buffers disagrees with the size computed by the LPA$SETIBF subroutine call.

1 = successful sweep started

nnn = VMS status code

4.5.4 LPA$DISWP — Initiate Synchronous Digital Input Sweep

The LPA$DISWP subroutine initiates digital input through a DR11-K. It is applicable in multirequest mode only.

The format of the LPA$DISWP subroutine call is as follows:

```
CALL LPA$DISWP (IBUF,LBUF,[NBUF],[MODE],[DWELL],[IEFN],-
     [LDELAY],[ICHN],[NCHN],[IND])
```

Arguments are as described in Section 4.5.1.2, with the following additions:

MODE Specifies sampling options. The VMS operating system 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 external triggers.)
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 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, the start method is digital input. If not selected, the start method is immediate. Multirequest 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.

IND Returns the success or failure status as follows:

0 = Error in call. Possible causes are the following: LPA$SETIBF subroutine was not previously called; LPA$RLSBUF subroutine was not previously called; size of data buffers disagrees with the size computed by the LPA$SETIBF subroutine call.

1 = successful sweep started

nnn = VMS status code

4.5.5 LPA$DOSWP — Initiate Synchronous Digital Output Sweep

The LPA$DOSWP subroutine initiates digital output through a DR11-K. It is applicable in multirequest mode only.

The format of the LPA$DOSWP subroutine call is as follows:

```
CALL LPA$DOSWP (IBUF,LBUF,[NBUF],[MODE],[DWELL],[IEFN],-
     [LDELAY],[ICHN],[NCHN],[IND])
```

Arguments are as described in Section 4.5.1.2, plus the following:

MODE Specifies sampling options. The VMS operating system 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 external triggers.)
4096	Start method. If selected, digital input start. If not selected, immediate start.
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 as follows:

0 = Error in call. Possible causes are the following: LPA$SETIBF subroutine was not previously called; LPA$RLSBUF subroutine was not previously called; size of data buffers disagrees with the size computed by the LPA$SETIBF subroutine call.

1 = successful sweep started

nnn = VMS status code

4.5.6 LPA$LAMSKS — Set LPA11-K Masks and NUM Buffer

The LPA$LAMSKS subroutine initializes a user buffer that contains a number to append to the logical name LPA11$, a digital start word mask, an event mark mask, and channel numbers for the two masks.

The LPA$LAMSKS subroutine must be called in the following cases:

- If users intend to use digital input starting or event marking
- If users 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 subroutine call is as follows:

```
CALL LPA$LAMSKS (LAMSKB,[NUM],[IUNIT],[IDSC],[IEMC],[IDSW],[IEMW],[IND])
```

Argument descriptions are as follows:

LAMSKB Specifies a four-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.

4.5.7 LPA$SETADC — Set Channel Information for Sweeps

The LPA$SETADC subroutine establishes channel start and increment information for the sweep control subroutines (see Table 4–4). It must be called to initialize IBUF before the LPA$SETADC subroutine is called.

The LPA$SETADC subroutine can be called in either of the following formats:

```
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 as follows: 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.

4.5.8 LPA$SETIBF — Set IBUF Array for Sweeps

The LPA$SETIBF subroutine initializes the IBUF array for use with the following subroutines:

LPA$ADSWP	LPA$DASWP	LPA$DISWP
LPA$DOSWP	LPA$IBFSTS	LPA$IGTBUF
LPA$INXTBF	LPA$IWTBUF	LPA$RLSBUF
LPA$RMVBUF	LPA$SETADC	LPA$STPSWP

The format of the LPA$SETIBF subroutine call is as follows:

```
CALL LPA$SETIBF (IBUF,[IND],[LAMSKB],BUF0,[BUF1,...,BUF7])
```

Arguments are as described in Section 4.5.1.2, with the following additions:

IBUF	Specifies a 50-longword array that is initialized by this subroutine. IBUF must be longword-aligned. (See Table 4–5 for additional information on IBUF.)
IND	Returns the success or failure status as follows: 0 = Error in call. Possible causes are the following: incorrect number of arguments; IBUF array not longword-aligned; buffer addresses not equidistant. 1 = IBUF initialized successfully
LAMSKB	Specifies the name of a four-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$LAMSKS subroutine call. (See Section 4.5.6 for a description of the LPA$LAMSKS subroutine.)
BUF0, . . .	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, the LPA$SETIBF subroutine verifies that all buffer addresses are equidistant. Buffers must be longword-aligned.

4.5.9 LPA$STPSWP — Stop In-Progress Sweep

The LPA$STPSWP subroutine allows you to stop a sweep that is in progress.

The format of the LPA$STPSWP subroutine call is as follows:

```
CALL LPA$STPSWP (IBUF,[IWHEN],[IND])
```

Arguments are as described in Section 4.5.1.2, with the following additions:

IBUF	The IBUF array specified in the LPA$ADSWP, LPA$DASWP, LPA$DISWP, or LPA$DOSWP subroutine call that initiated the sweep.
IWHEN	Specifies when to stop the sweep. The VMS operating system 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 VMS format: 1 = Success nnn = VMS error code issued by the $CANCEL system service

Note that when the LPA$STPSWP subroutine is returned, the sweep cannot be stopped. If it is necessary to wait until the sweep has stopped, you can call the LPA$IWTBUF subroutine in a loop until it returns IBUFNO = –1 (see Section 4.5.16).

4.5.10 LPA$CLOCKA — Clock A Control

The LPA$CLOCKA subroutine sets the clock rate for Clock A.

The format of the LPA$CLOCKA subroutine call is as follows:

```
CALL LPA$CLOCKA (IRATE,IPRSET,[IND],[NUM])
```

Arguments are as described in Section 4.5.1.2, with the following additions:

IRATE — Specifies the clock rate. One of the following values must be specified:

Value	Meaning
–1	Direct-coupled Schmidt trigger 1. Used only for A/D sweeps in dedicated mode, that is, MODE = 512 (see Section 4.5.2).
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 = VMS error code indicating an I/O error

NUM — Specifies the number to be appended to the logical name LPA11$. The default value is 0. This subroutine sets Clock A on the LPA11-K assigned to LPA11$num.

4.5.11 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 subroutine call is as follows:

```
CALL LPA$CLOCKB ([IRATE],IPRSET,MODE,[IND],[NUM])
```

Arguments are as described in Section 4.5.1.2, with the following additions:

IRATE — Specifies the clock rate. One of the following must be specified:

Value	Meaning
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 eight 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. The VMS operating system defines the following:

1 = Clock B operates in noninterrupt mode.

2 = The feed B to A bit in the Clock B status register will be set (see the *LPA11-K Laboratory Peripheral Accelerator User's Guide*).

IND — Receives a success or failure code as follows:

1 = Clock B set successfully

nnn = VMS error code indicating an I/O error

NUM — Specifies the number to be appended to the logical name LPA11$. The default value is 0. This subroutine sets Clock B on the LPA11-K assigned to LPA11$num.

4.5.12 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 LPA$XRATE subroutine can be called in either of the following formats:

```
CALL LPA$XRATE (AINTRVL,IRATE,IPRSET,IFLAG)

ACTUAL=LPA$XRATE(AINTRVL,IRATE,IPRSET,IFLAG)
```

Arguments are as described in Section 4.5.1.2, 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 round-off errors, the resulting intersample time can be different from the specified intersample time. Note that when the LPA$XRATE subroutine is called from VAX FORTRAN IV-PLUS programs as a function, it must be explicitly declared a real function. Otherwise, the LPA$XRATE subroutine defaults to an integer function.

If AINTRVL is either too large or too small to be achieved, both IRATE and ACTUAL are returned to 0.

4.5.13 LPA$IBFSTS — Return Buffer Status

The LPA$IBFSTS subroutine returns information on the buffers used in a sweep.

The format of the LPA$IBFSTS subroutine 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 it is the next to be filled or emptied by the LPA11-K.

4.5.14 LPA$IGTBUF — Return Buffer Number

The LPA$IGTBUF subroutine returns the number of the next buffer to be processed by the application program, the buffer at the head of the user queue (see Figure 4–3). It 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, the LPA$IWTBUF, not the LPA$IGTBUF subroutine, should be called.

The LPA$IGTBUF subroutine can be called in one of these formats:

```
CALL LPA$IGTBUF (IBUF,IBUFNO)

IBUFNO=LPA$IGTBUF(IBUF)
```

Arguments are as described in Section 4.5.1.2, plus the following:

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 4–6 lists the possible combinations of IBUFNO and IOSB contents on the return from a call to the LPA$IGTBUF subroutine. The first four words of the IBUF array contain the I/O status block (IOSB). If IBUFNO is –1, the IOSB must be checked to determine the reason.

Table 4–6 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	VMS error code	0	LPA11-K ready-out and maintenance registers (only if SS$DEVREQERR, SS$_CTRLERR, or SS$DEVCMDERR is returned)	No buffers in queue. Sweep terminated due to error condition. Section 4.6 describes the VMS error codes; the *LPA11-K Laboratory Peripheral Accelerator User's Guide* lists the LPA11-K error codes.

4.5.15 LPA$INXTBF — Set Next Buffer to Use

The LPA$INXTBF subroutine alters the normal buffer selection algorithm so that you can specify the next buffer to be filled or emptied. The specified buffer is reinserted at the head of the device queue.

The LPA$INXTBF subroutine can be called in one of these formats:

```
CALL LPA$INXTBF (IBUF,IBUFNO,IND)

IND=LPA$INXTBF(IBUF,IBUFNO)
```

Arguments are as described in Section 4.5.1.2, plus the following:

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 as follows: 0 = Specified buffer not in the device queue 1 = Next buffer successfully set

4.5.16 LPA$IWTBUF — Return Next Buffer or Wait

The LPA$IWTBUF subroutine returns the next buffer to be processed by the application program, the buffer at the head of the user queue. If the user queue is empty, the LPA$IWTBUF subroutine 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 LPA$IWTBUF subroutine can be called in either of the following formats:

```
CALL LPA$IWTBUF (IBUF,[IEFN],IBUFNO)

IBUFNO=LPA$IWTBUF(IBUF,[IEFN])
```

Arguments are as described in Section 4.5.1.2, with the following additions:

IBUF	The IBUF array specified in the call that initiated the sweep.
IEFN	Not used. This argument provides compatibility with the operating system. (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 4–7 lists the possible combinations of IBUFNO and I/O status block contents on the return from a call to the LPA$IWTBUF subroutine. The first four words of the IBUF array contain the I/O status block. If IBUFNO is –1, the I/O status block must be checked to determine the reason.

Table 4–7 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	VMS error code	0	LPA11-K ready-out and maintenance registers (only if SS$_DEVREQERR SS$_CTRLERR, or SS$_ DEVCMDERR is returned)	No buffers in queue. Sweep terminated due to error condition. Section 4.6 describes the VMS error codes; the *LPA11-K Laboratory Peripheral Accelerator User's Guide* lists the LPA11-K error codes.

4.5.17 LPA$RLSBUF — Release Data Buffer

The LPA$RLSBUF subroutine declares one or more buffers available to be filled or emptied by the LPA11-K. It inserts the buffer at the tail of the device queue (see Figure 4–3).

The format of the LPA$RLSBUF subroutine call is as follows:

```
CALL LPA$RLSBUF (IBUF,[IND],INDEX0,INDEX1,...,INDEXN)
```

Arguments are as described in Section 4.5.1.2, with the following additions:

IBUF	The IBUF array specified in the call that initiated the sweep.
IND	Returns the success or failure status as follows: 0 = Buffer number was illegal, the number of arguments specified was incomplete, or a double buffer overrun occurred. A double buffer overrun can occur only 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). 1 = Buffer(s) released successfully.
INDEX0, . . .	Specify the indexes (0-7) of the buffers to be released. A maximum of eight indexes can be specified.

The LPA$RLSBUF subroutine must be called to release a buffer (or buffers) to the device queue before the sweep is initiated. (See Section 4.5.1.1 for a discussion of buffer management.) Note that the LPA$RLSBUF subroutine does not verify whether the specified buffers are already in a queue. If a buffer is released when it is already in a queue, the queue pointers are invalidated and unpredictable results can occur.

If buffer overrun is specified as a nonfatal error, buffer 0 should not be released before the sweep is initiated. However, if either the LPA$IGTBUF or LPA$IWTBUF subroutine returns buffer 0, it should be released. 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 subroutine returns an error the next time buffer 0 is released.

4.5.18 LPA$RMVBUF — Remove Buffer from Device Queue

The LPA$RMVBUF subroutine removes a buffer from the device queue.

The format of the LPA$RMVBUF subroutine call is as follows:

```
CALL LPA$RMVBUF (IBUF,IBUFNO,[IND])
```

Arguments are as described in Section 4.5.1.2, 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 as follows: 0 = Buffer not found in the device queue 1 = Buffer successfully removed from the device queue

4.5.19 LPA$CVADF — Convert A/D Input to Floating-Point

The LPA$CVADF subroutine converts A/D input values to floating-point numbers. It is supported to provide compatibility with the VMS operating system.

The LPA$CVADF subroutine can be called in either of the following formats:

```
CALL LPA$CVADF (IVAL,VAL)
VAL=LPA$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.

4.5.20 LPA$FLT16 — Convert Unsigned 16-Bit Integer to Floating-Point

The LPA$FLP16 subroutine converts unsigned 16-bit integers to floating point. It is supported to provide compatibility with the VMS operating system.

The LPA$FLT16 subroutine can be called in either of the following formats:

```
CALL LPA$FLT16 (IVAL,VAL)
VAL=LPA$FLT16(IVAL)
```

Argument descriptions are as follows:

IVAL	An unsigned 16-bit integer.
VAL	Receives the converted value.

4.5.21 LPA$LOADMC — Load Microcode and Initialize LPA11-K

The LPA$LOADMC subroutine provides a program interface to the LPA11-K microcode loader. It sends a load request through a mailbox to the loader process to load microcode and to initialize an LPA11-K. (Section 4.7.1 describes the microcode loader process.)

The format of the LPA$LOADMC subroutine call is as follows:

```
CALL LPA$LOADMC ([ITYPE][,NUM][,IND][,IERROR])
```

Argument descriptions are as follows:

ITYPE — The type of microcode to be loaded. The VMS operating system defines the following values:

Value	Meaning
1	Multirequest mode; default value
2	Dedicated A/D mode
3	Dedicated D/A mode

NUM — The number to be appended to the logical name LPA11$. The default value is 0.

IND — Receives the completion status as follows:

1 = Microcode loaded successfully

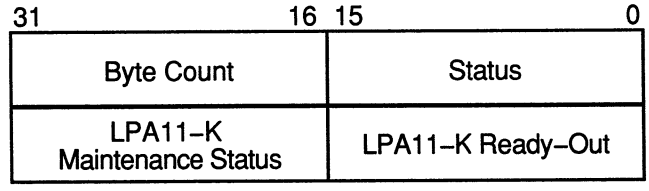
nnn = VMS error code

IERROR — Provides additional error information. Receives the second longword of the I/O status block if SS$_CTRLERR, SS$_DEVCMDERR, or SS$_DEVREQERR is returned in IND. Otherwise, the contents of IERROR are undefined.

4.6 I/O Status Block

The I/O status block (IOSB) format for the load microcode, start microprocessor, initialize LPA11-K, set clock, and start data transfer request QIO functions is shown in Figure 4–4.

Figure 4–4 I/O Functions IOSB Content



ZK–0662–GE

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 the *LPA11-K Laboratory Peripheral Accelerator User's Guide*). Appendix A of this manual lists the status returns for LPA11-K I/O functions. (The *VMS System Messages*

and Recovery Procedures Reference Manual provides explanations and suggested user actions for these returns.)

If high-level language library procedures are used, the status returns listed in Appendix A 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 I/O status block for the high-level language procedures.

4.7 Loading LPA11-K Microcode

The microcode loading and device initialization routines automatically load microcode during system initialization (if specified in the system manager's startup file) and during power recovery. These routines also allow a nonprivileged user to load microcode and to restart the system.

The LPA11-K loader and initialization routines consist of the following parts:

- A microcode loader process that 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 that accepts operator commands or indirect file commands to load microcode and to 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 to initialize an LPA11-K. Section 4.5.21 describes that routine in greater detail.

4.7.1 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 bit map 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 4–1) 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 bit map 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

4.7.2 Operator Process

The operator process loads microcode and initializes an LPA11-K through either terminal or indirect file commands. To run the operator process, type RUN SYS$SYSTEM:LALOAD. The command input syntax is as follows:

```
devname/type
```

In the preceding example, *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 the following types of microcode to load:

- /MULTI_REQUEST—Multirequest mode
- /ANALOG_DIGITAL—Dedicated A/D mode
- /DIGITAL_ANALOG—Dedicated D/A mode

If */type* 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.

4.8 RSX–11M/M–PLUS and VMS Differences

This section lists those areas of the VMS high-level language support routines that differ from the RSX–11M LPA11-K routines. The *RSX–11M/M–PLUS I/O Drivers Reference Manual* provides a detailed description of the RSX–11M LPA11-K support routines. Differences between the VMS and RSX–11M/M–PLUS routines can be determined by comparing the descriptions in the *RSX–11M/M–PLUS I/O Drivers Reference Manual* with the descriptions for the VMS routines in the preceding sections of this chapter.

4.8.1 General

The following are general features of VMS high-level support routines:

- The LUN argument is not used. 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 (IOSB) values returned by the LPA$IWTBUF and LPA$IGTBUF subroutines are different.

4.8.2 Alignment and Length

The following are features of alignment and length in VMS high-level support routines:

- Buffers must be contiguous.
- Buffers must be longword-aligned.
- The random channel list (RCL) must be word-aligned.
- The IBUF array length is 50 longwords and must be longword-aligned.

4.8.3 Status Returns

The following are features of status returns in VMS high-level support routines:

- The I/O status block (IOSB) length is eight bytes; numeric values of errors differ.
- Several routines return the following:

  1 = Success

  0 = Failure detected in support routine

  nnn = VMS status code; failure detected in system service

4.8.4 Sweep Routines

The following are features of sweep routines in VMS high-level support routines:

- If an event flag is specified, it must be within a %VAL( ) construction.
- A tenth argument, IND, is added to return the success or failure status.

4.9 LPA11-K Programming Examples

The following programming examples use LPA11-K high-level language procedures and LPA11-K Queue I/O functions.

The *VMS Device Support Manual* volume contains information that is applicable to LPA11-K programming.

4.9.1 LPA11-K High-Level Language Program (Program A)

This sample program (Example 4–1) 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 4–8 lists the variables used in this program.

Table 4–8 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.

Example 4–1 LPA11-K High-Level Language Program (Program A)

```
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))
```

(continued on next page)

Example 4–1 (Cont.) LPA11-K High-Level Language Program (Program A)

```
C
C  Set clock rate to 1 khz, clock preset to -10.
C
        CALL LPA$CLOCKA(4,-10,ISTAT)
        IF (.NOT. ISTAT) GO TO 950
C
C  Initialize IBUF array for sweep.
C
        CALL LPA$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$IWTBUF(IBUF)
        IF (BUFNUM .LT. 0) GO TO 800
C
C  Process data in buffer (1,BUFNUM) to buffer (1000,BUFNUM).
        .
        .
        .
(Application-dependent code is inserted at this point.)
        .
        .
        .
C  Release buffer is filled again.
C
200     CALL LPA$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  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 a
C  VMS error code.
C
```

(continued on next page)

Example 4–1 (Cont.) LPA11-K High-Level Language Program (Program A)

```
950     IF (ISTAT .NE. 0) CALL LIB$STOP(%VAL(ISTAT))
        PRINT   *,'ERROR IN LPA11-K SUBROUTINE CALL'
2000    STOP
        END

C  ****************************************************************
```

4.9.2 LPA11-K High-Level Language Program (Program B)

This program (Example 4–2) 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 nonfatal error)
- Random channel list (RCL) addressing
- Sequential channel addressing

Table 4–9 lists the variables used in this program.

Table 4–9 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 that contains the random channel list (RCL)
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

Example 4–2 LPA11-K High-Level Language Program (Program B)

```
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
```

(continued on next page)

Example 4–2 (Cont.) LPA11-K High-Level Language Program (Program B)

```
        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 nonfatal.
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).
        .
        .
        .
```

(continued on next page)

Example 4–2 (Cont.) LPA11-K High-Level Language Program (Program B)

```
C
C  Release buffers for D/A sweep.
C
        CALL LPA$RLSBUF (IBUFDA,ISTAT,0,1)
        IF (.NOT. ISTAT) GO TO 5000
C  ****************************************************************
C
C  Start both sweeps
C
C  ****************************************************************
C
C  Start A/D sweep. Mode bits specify buffer overrun is nonfatal and
C  multirequest mode. Sweep arguments specify 500 samples/buffer,
C  Indefinite sampling, dwell = 10 clock overflows, synchronize using
C  event flag 15, and a delay of 50 clock overflows.
C
        MODE = 16384 + 64
        CALL LPA$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$DASWP(IBUFDA,2000,15,MODE,25,FILLBF,10,,,ISTAT)
        (.NOT.  ISTAT) GO TO 5000
C  ****************************************************************
C
C  Wait for an A/D buffer and then process the data it contains.  D/A
C  buffers are filled asynchronously by the completion routine FILLBF.
C
C  ****************************************************************
C
C  Wait for a buffer to be filled by A/D.  If BUFNUM is less than
C  zero, the sweep has stopped (either successfully or with an error).
C
100     BUFNUM = LPA$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.)
        .
        .
        .
```

(continued on next page)

Example 4–2 (Cont.) LPA11-K High-Level Language Program (Program B)

```
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 is 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 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
```

(continued on next page)

Example 4–2 (Cont.) LPA11-K High-Level Language Program (Program B)

```
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  ****************************************************************
```

4.9.3 LPA11-K QIO Functions Program (Program C)

This sample program (Example 4–3) 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.

Example 4–3 LPA11-K QIO Functions Program (Program C)

```
; ****************************************************************
;
;                              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,
                                        ; multirequest 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
```

(continued on next page)

Example 4–3 (Cont.) LPA11-K QIO Functions Program (Program C)

```
DEVNAME: .ASCID /LAA0/

CHANNEL: .BLKW  1                       ; Contains channel number

        .PSECT  LACODE,NOWRT

START:  .ENTRY  START,^m<>
        $ASSIGN_S DEVNAM=DEVNAME,CHAN=CHANNEL ; Assign channel
        BLBS    R0,5$                   ; No error
        BRW     ERROR                   ; Error

5$:                                     ; Set clock overflow rate
                                        ; To 2 khz.  (1 mhz rate
                                        ; divided by 500 preset)
        $QIOW_S ,CHAN=CHANNEL,FUNC=#IO$_SETCLOCK,-
                IOSB=IOSB,,,,P2=#1,P3=#^X143,P4#-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=IOSB,,,P1=CBUFF,P2=#40,P3=#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,G^LIB$STOP           ; Signal error

BFRAST: BFRAST,m^<>                     ; 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

; ***************************************************************
```

5 Line Printer Driver

This chapter describes the use of the line printer drivers LPDRIVER and LCDRIVER.

5.1 Supported Line Printer Devices

The following sections describe the line printer controllers and line printers supported by the VMS operating system.

5.1.1 LP11 Line Printer Controller

The LP11 line printer controller provides an interface between the VAX UNIBUS adapter and the line printer. The LP11 performs the following functions:

- Synchronizes single-character data transfers from the UNIBUS to the printer
- Informs the VMS operating system about printer status
- Enables the printer to gain control of the UNIBUS to report interrupts

5.1.2 DMF32 and DMB32 Line Printer Controllers

The DMF32 and DMB32 line printer controllers provide a direct memory access (DMA) interface between the VAX UNIBUS adapter (for the DMF32), or the VAXBI adapter (for the DMB32), and the line printer. The DMF32/DMB32 optionally perform the following functions:

- Tab expansion
- Carriage control
- Line wrapping and truncation
- Case conversion
- Passall mode
- Printall mode

5.1.3 LP27 Line Printer

The LP27 line printer is a high-speed, 132-column line printer, available with either a 64- or 96-character ASCII print set. The LP27-U is a fully buffered model that operates at a standard speed of up to 1200 lines per minute. Forms with up to six parts can be used for multiple copies. A version of the LP27 is available for operation of the printer up to 24.5 meters (1000 feet) from the host.

5.1.4 LA11 DECprinter I

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

5.1.5 LN01 Laser Page Printer

The LN01 laser page printer is a nonimpact printer that employs laser technology to produce high-quality print. Using electrophotographic imaging and xerographic printing, the LN01 prints one page at a time at a rate of 12 pages per minute. The print resolution of 300 x 300 dots per square inch produces characters of even density and alignment. The LN01 uses two, 188-character, fixed-space fonts; the column width is 132 characters.

5.1.6 LN03 Laser Page Printer

The LN03 laser page printer is a table-top, nonimpact page printer that uses laser imaging and xerographic printing techniques. The LN03 has a printing speed of 8 pages per minute with a print resolution of 300 x 300 dots per square inch. Four built-in fonts are available. Several column widths, including 80 or 132 characters, are also available.

5.2 Driver Features

The line printer drivers provide output character formatting and error recovery. These features are 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 unless the LP$M_TAB characteristic is clear.
- All lowercase alphabetic characters are converted to uppercase before printing (unless the characteristic specifying lowercase characters is set; see Section 5.4.3 and Table 5–2).
- On printers where the line-feed, form-feed, vertical-tab, and carriage-return characters empty the printer buffer, returns are held back and output only if the next character is not a form feed, line feed, or vertical tab. Carriage returns are always output on units that have the LP$M_CR characteristic set (see Section 5.4.3 and Table 5–2).

- The horizontal print position is incremented on the output of all characters, including the space character. Characters are discarded if the horizontal print position is equal to or greater than the carriage width, unless the LP$M_WRAP characteristic is set or the LP$M_TRUNCATE characteristic is clear (see Section 5.3).
- On printers without a 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 VMS line printer drivers perform the following error recovery operations:

- If the printer is off line 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 after 30 seconds. Successive messages, if they occur, are sent 1, 2, 4, 8, . . . minutes after the initial message.
- The current operation is retried every two seconds to test for a changed situation, such as the printer coming on line.
- The current I/O operation can be canceled at the next timeout without the printer being on line.
- When the printer comes on line, device operation resumes automatically.

5.3 Line Printer Driver Device Information

You can obtain information on printer characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *VMS System Services Reference Manual.*)

$GETDVI returns line printer characteristics when you specify the item codes DVI$_DEVCHAR and DVI$_DEVDEPEND. Tables 5–1 and 5–2 list these characteristics. The $DEVDEF macro defines the device-independent characteristics; the $LPDEF macro defines the device-dependent characteristics. DVI$_DEVDEPEND returns a longword field that contains the device-dependent characteristics in the three low-order bytes and the page length in the high-order byte. Maximum page length is 255.

DVI$_DEVTYPE and DVI$_DEVCLASS return the device type and class names, which are defined by the $DCDEF macro. The device type is a value that corresponds to the printer, for example, LP$_LP27 or LP$_LA11. The device class for printers is DC$_LP. DVI$_DEVBUFSIZ returns

the page width, which is a value in the range of 0 through 255 on a DMF32 controller and 0 through 65535 on an LP11 or a DMB32 controller.

Table 5–1 Printer Device-Independent Characteristics

Characteristic[1]	Meaning
	Dynamic Bits (Conditionally Set)
DEV$M_SPL	Device is spooled.
DEV$M_AVL	Printer is on line and available.
	Static Bits (Always Set)
DEV$M_REC	Device is record-oriented.
DEV$M_CCL	Carriage control is enabled.
DEV$M_ODV	Device is capable of output.

[1]Defined by the $DEVDEF macro.

Table 5–2 Device-Dependent Characteristics for Line Printers

Value[1]	Meaning
LP$M_CR	Printer requires carriage return. (See Section 5.2.1).
LP$M_FALLBACK	Printer translates multinational characters to a seven-bit equivalent representation if possible. Otherwise, an underscore character (_) replaces the character. LPM$M_FALLBACK has no effect on physical block operations. See Appendix B for a list of multinational characters.
LP$M_LOWER	Printer can print lowercase characters. If this value is not set, all lowercase characters are converted to uppercase when output. (LP$M_LOWER has no effect on write physical block operations.)
LP$M_MECHFORM	Printer has mechanical form feed. This characteristic is used when variable form length is required, such as in 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$M_PASSALL	All output data is in binary (no data interpretation occurs). Data termination occurs when the buffer is full (default buffer size is 132 bytes). Character formatting is disabled.
LP$M_PRINTALL	All printing and nonprinting characters are transferred to the printer, while character formatting remains enabled.
LP$M_TAB	Printer enables tab expansion.
LP$M_TRUNCATE	Printer truncates records that are larger than the carriage width.

[1]Defined by the $LPDEF macro.

(continued on next page)

Table 5–2 (Cont.) Device-Dependent Characteristics for Line Printers

Value[1]	Meaning
LP$M_WRAP	Printer wraps records that are larger than the carriage width. If a string of text is longer than the width specified in the second longword, the string is continued on the next line.

[1] Defined by the $LPDEF macro.

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 take 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- or 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
- 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. The default buffer size is 132 bytes.

If the printer is not set spooled, write virtual block and write logical block operations perform the same function. If the printer is set spooled, a write logical block function queues the I/O to the printer, and a write virtual block 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 printer do not include LP$M_LOWER. (This does not apply to write physical block operations.)

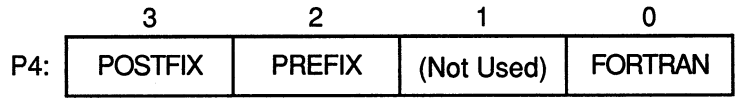
Multiple line feeds are generated for form feeds only if the printer does not have a mechanical form feed (LP$M_MECHFORM) characteristic. The number of line feeds generated depends on the current page position and the page length.

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 5–1 shows the P4 longword format.

Figure 5–1 P4 Carriage Control Specifier



ZK-0664-GE

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 as follows:

1 Prefix carriage control

2 Print

3 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: carriage-return/line-feed combination[1], print buffer contents, return)
30	0	Double-space carriage control (Sequence: carriage-return/line-feed combination, carriage-return/line-feed combination, print buffer contents, return)
31	1	Page eject carriage control (Sequence: form feed, print buffer contents, return)
2B	+	Overprint carriage control; allows double printing for emphasis or for special effects (Sequence: print buffer contents, return)
24	$	Prompt carriage control (Sequence: carriage-return/line-feed combination, print buffer contents)
All other values		Same as ASCII space character: single-space carriage control

[1]A carriage-return/line-feed combination is a carriage return followed by a line feed.

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 carriage-return/line-feed combinations.

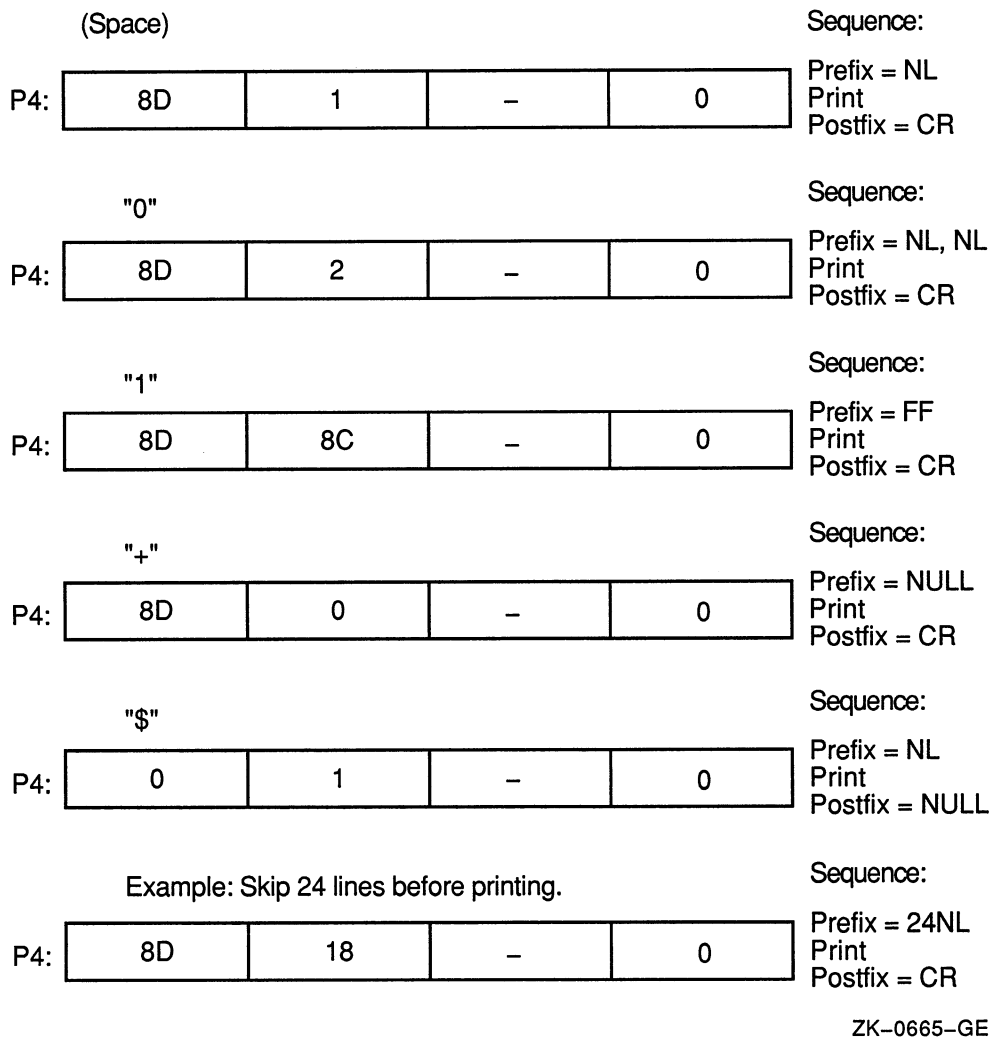
Bit 7	Bit 6	Bit 5	Bits 0–4	Meaning
1	0	0	1–1F	Output the single ASCII control character specified by the configuration of bits 0 through 4 (seven-bit character set).
1	1	0	1–1F	Output the single ASCII control character specified by the configuration of bits 0 through 4, which are translated as ASCII characters 128 through 159 (eight-bit character set; see Appendix B).

Figure 5–2 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.

In the first example, the prefix/postfix hexadecimal coding for a single-space carriage control (carriage-return/line-feed combination, print buffer contents, carriage-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)
```

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



5.4.2 Sense Printer Mode

The sense printer mode function senses the current device-dependent printer characteristics and returns them in the second longword of the I/O status block. No device- or 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. The VMS operating system 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. The following function codes are provided:

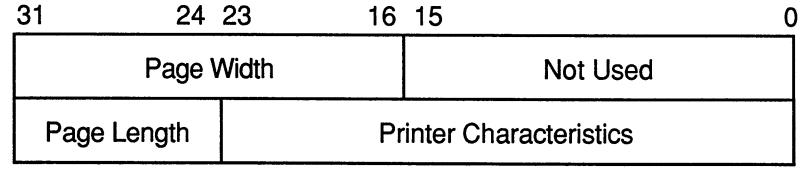
- IO$_SETMODE
- IO$_SETCHAR

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

P1—The address of a characteristics buffer

Figure 5–3 shows the quadword P1 characteristics buffer for IO$_SETMODE. Figure 5–4 shows the same buffer for IO$_SETCHAR.

Figure 5–3 Set Mode Buffer

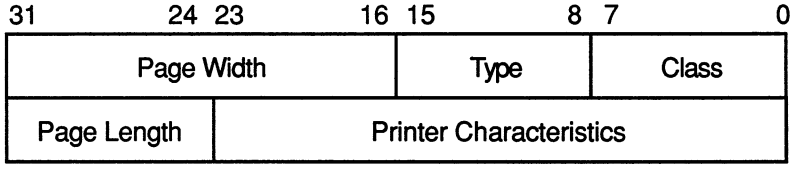


ZK–0666–GE

In the buffer, the device class is DC$_LP. The printer type is a value that corresponds to the printer: DT$_LP27 or DT$_LA11. The type can be changed by the IO$_SETCHAR function. The page width is a value in the range of 0 through 255 on a DMF32 controller and 0 through 65535 on an LP11 or DMB32 controller.

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

Figure 5–4 Set Characteristics Buffer



Application programs that change specific line printer characteristics should perform the following steps:

1 Use the IO$_SENSEMODE function to read the current characteristics.

2 Modify the characteristics.

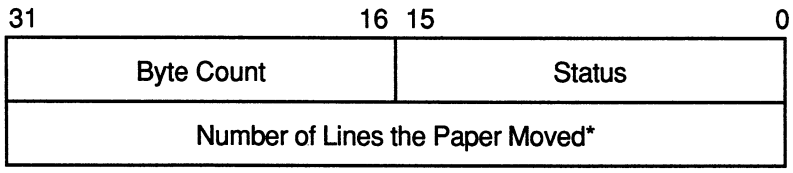
3 Use the set mode function to write back the results.

Failure to follow this sequence will result in clearing any previously set characteristic.

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 5–5 and 5–6. Appendix A lists the status returns for these functions. (The *VMS System Messages and Recovery Procedures Reference Manual* provides explanations and suggested user actions for these returns.)

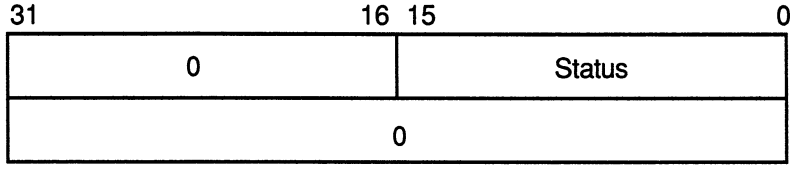
Figure 5–5 IOSB Contents — Write Function



* 0 if IO$_WRITEPBLK

ZK–0668–GE

Figure 5–6 IOSB Contents — Set Mode Function



ZK–0669–GE

5.6 Line Printer Driver Programming Example

The following sample program (Example 5–1) 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.

5.6 Line Printer Driver Programming Example

Example 5–1 Line Printer Program Example

```
; ****************************************************************
;
        .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 set up 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
```

(continued on next page)

Example 5–1 (Cont.) Line Printer Program Example

```
;
; ****************************************************************
;
;                       Start Program
;
; ****************************************************************

;
; 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 ($QIOW) system service requests.  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 = 0, assign failure
        MOVL    #11,R3                  ;Set up loop count
        MOVAL   OUTPUT_FORMAT,R2        ;Set up o/p format address
                                        ;in R2
;
; Start the 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 autoincrement
        BLBC    R0,40$                  ;If low bit = 0, I/O failure
        SOBGTR  R3,30$                  ;Branch if not finished
40$:    $DASSGN_S CHAN=DEVICE_CHANNEL   ;Deassign channel
50$:    RET                             ;Return

        .END    PRINTER_EXAMPLE
```

6 Magnetic Tape Drivers

This chapter describes the use of the VMS magnetic tape drivers, drives, and controllers. These drivers support the devices listed in Table 6–1.

Table 6–1 Supported Magnetic Tape Devices

Drive[1]	Code	No. of Tracks	Recording Density (bpi) 800	1600	6250	Tape Speed (ips)	Max. Data Transfer Rate (bps)	Recording Method[2]
HSC-Series Controllers								
TA78	MU	9	No	Yes	Yes	125	200,000 (1600 bpi) 781,250 (6250 bpi)	PE or GCR
TA79[3]	MU	9	No	Yes	Yes	125	769,000	PE or GCR
TA81	MU	9	No	Yes	Yes	25 or 75	120,000 (1600 bpi) 468,750 (6250 bpi)	PE or GCR
TA90[3]	MU	18	No	No	38,000	79	2,100,000[6]	X3B5
LESI Adapters								
RV20	MU	9	No	No	Yes	N/A	1.33 MB	Write-once optical disk
TQ81	MU	9	No	Yes	Yes	25 or 75	120,000 (1600 bpi) 468,750 (6250 bpi)	PE or GCR
TU81[3]	MU	9	No	Yes	Yes	25 or 75	120,000 (1600 bpi) 468,750 (6250 bpi)	PE or GCR
TU81-Plus	MU	9	No	Yes	Yes	25 or 75	120,000 (1600 bpi) 468,750 (6250 bpi)	PE or GCR
TM03 Controller								
TE16	MT	9	Yes	Yes	No	45	36,000 (800 bpi) 72,000 (1600 bpi)	NRZI or PE
TU45	MT	9	Yes	Yes	No	75	60,000 (800 bpi) 120,000 (1600 bpi)	NRZI or PE

[1]The RV20, TK70, TQK50, TUK50, TU81, TU81-Plus, TA78, TU78, TA81, and TA90 are tape mass storage control protocol (TMSCP) drives.

[2]NRZI = non-return-to-zero-inverted; PE = phase encoded; GCR = group-coded recording; MFM = modified frequency modulation; HDMFM = high density modified frequency modulation; X3B5 = format adheres to the proposed ANSI standard X3B5.

[3]Has a self-contained controller.

[6]Dependent upon host system capabilities. Has a data collection and transfer rates of 2.1 MB per second.

(continued on next page)

Table 6–1 (Cont.) Supported Magnetic Tape Devices

Drive[1]	Code	No. of Tracks	Recording Density (bpi) 800	1600	6250	Tape Speed (ips)	Max. Data Transfer Rate (bps)	Recording Method[2]
TM03 Controller								
TU77	MT	9	Yes	Yes	No	125	100,000 (800 bpi) 200,000 (1600 bpi)	NRZI or PE
TM78 Controller								
TU78	MF	9	No	Yes	Yes	125	200,000 (1600 bpi) 781,250 (6250 bpi)	PE or GCR
TM79 Controller								
TU79	MF	9	No	Yes	Yes	125	769,000	PE or GCR
TBK70 and TQK70 Controllers								
TK70	MU	48	No	No	10000	100	90,000	HDMFM
TUK50, TQK50, and TZK50 Controllers								
TK50[5]	MU	22[4]	No	No	6666	75	45,000	MFM
TZ30[3]	MU	22[4]	No	No	6666	75	45,000	MFM
TS11 Controller								
TS04	MS	9	No	Yes	No	45	72,000	PE
TS05	MS	9	No	Yes	No	25	40,000	PE
TU80[3]	MS	9	No	Yes	No	25 or 100	160,000	PE

[1]The RV20, TK70, TQK50, TUK50, TU81, TU81-Plus, TA78, TU78, TA81, and TA90 are tape mass storage control protocol (TMSCP) drives.

[2]NRZI = non-return-to-zero-inverted; PE = phase encoded; GCR = group-coded recording; MFM = modified frequency modulation; HDMFM = high density modified frequency modulation; X3B5 = format adheres to the proposed ANSI standard X3B5.

[3]Has a self-contained controller.

[4]Each track written separately—not in parallel.

[5]The TK50 is a tape mass storage control protocol (TMSCP) device when configured on (. . . BA23 BA123 w/s) systems. The TK50 has a self-contained controller when configured on VAXstation 2000 and MicroVAX 2000 systems.

6.1 Supported Magnetic Tape Controllers

The following sections describe the VMS magnetic tape controllers.

6.1.1 TM03 Magnetic Tape Controller

The TM03 magnetic tape controller supports up to eight TE16, TU45, or TU77 tape drives. These dual-density (800 or 1600 bpi) drives differ in speed: the TE16, TU45, and TU77 read and write data at 45, 75, and 125 inches per second, respectively. Each drive can hold one 2400-foot, 9-track reel with a capacity of approximately 40 million characters. The TM03 controller is connected to the MASSBUS through a MASSBUS adapter.

6.1.2 TS11 Magnetic Tape Controller

The TS11 magnetic tape controller connects to the UNIBUS through a UNIBUS adapter and supports one TS04 tape drive. The TS11/TS04 is a single-density tape system that supports 1600-bpi, phase-encoded recording.

The TSU05 and the TSV05 magnetic tape drives are used with UNIBUS and Q-bus systems, respectively.

6.1.3 TM78 and TM79 Magnetic Tape Controllers

The TM78 and TM79 magnetic tape controllers support up to four TU78 tape drives. These high-performance, dual-density drives (1600 or 6250 bpi) operate at 125 inches per second (ips) using a 2400-foot reel of tape with a capacity of approximately 146 million characters when recorded in the GCR (6250 bpi) mode. The TM78 and TM79 controllers are connected to the MASSBUS through a MASSBUS adapter.

6.1.4 TU80 Magnetic Tape Subsystem

The TU80 is a single-density, dual-speed (25 or 100 ips) magnetic tape subsystem that uses streaming tape technology (see Section 6.2.7). It supports one drive per subsystem. The TU80 connects to the UNIBUS through a UNIBUS adapter and completely emulates the TS11 magnetic tape controller.

6.1.5 TU81 and TA81 Magnetic Tape Subsystems

The TU81 and the TA81 are high-performance, dual-density (1600 or 6250 bpi), dual-speed (25 or 75 ips) magnetic tape subsystems that use streaming tape technology (see Section 6.2.7). The TU81 connects to the UNIBUS through a UNIBUS adapter. The TA81 attaches to an HSC50 controller. Both drives are managed with the tape mass storage control protocol (TMSCP).

6.1.6 TU81-Plus Magnetic Tape Subsystem

The TU81-PLUS is an enhanced version of the TU81 streaming tape subsystem. It is a 9-track, dual-speed, dual-density, ANSI-standard, half-inch magnetic tape subsystem. In addition, it has a 256-kilobyte (KB) cache buffer that temporarily stores commands and data moving to and from the tape unit. The buffer increases the amount of time the tape drive is able to stream, thereby increasing performance. The TU81-PLUS connects to all VAXBI, UNIBUS, and Q-bus systems using the KLESI-B, KLESI-U, and KLESI-Q adapters.

6.1.7 TA90 Magnetic Tape Subsystem

The TA90 is a 5- by 4-inch, 200-MB cartridge tape, fully read- and write-compatible with the IBM® 3480 format. The TA90 includes a master controller and a dual transport unit. As many as three additional dual transport slave units can be connected to a single TA90 master controller for a total of eight drives. The controller connects to the HSC 5X-DA high-speed channel card in the HSC.

TA90 tape drives can be equipped with optional stack loaders for unattended backup operations. Each TA90 master has two dual-port STI connections to the HSC. Such dual pathing allows each control unit to service two HSC controllers which significantly increases tape drive availability. The TA90 subsystem includes a 2-MB cache which allows the controller to prefetch upcoming commands and store them while completing current data transfers. This behavior helps optimize performance. The TA90 is a TMSCP device.

6.1.8 RV20 Write-Once Optical Drive

The RV20, a 2-gigabyte, double-sided, write-once optical (WORM) disk drive is accessed sequentially similar to a tape. A 100-bit error correction code (ECC) protects user data. The controller performs bad block replacement. Three RV20 slaves can be daisy-chained to the subsystem controller in the RV20 master for a total of four drives.

RV02 cartridges can be used on any DIGITAL RV20 optical subsystem.

The average access time is 212.5 ms with an average seek rate of 150 ms. The maximum data transfer rate is 262 KB per second (formatted and sustained) with a burst rate of 1.33 MB per second.

6.1.9 TK50 Cartridge Tape System

The TK50 is a 95-MB, 5.25-inch cartridge tape system that uses streaming tape technology (see Section 6.2.7). The TK50 records data serially on 22 tracks using serpentine recording, rather than on separate (parallel) tracks. Data written to tape is automatically read as it is written. A CRC check is performed and the controller is notified immediately if an error occurs on the tape.

The TQK50 is a dual-height Q-bus controller for the TK50 tape drive. The TUK50 is a UNIBUS controller for the same drive. The TZK50 is a SCSI controller for the TK50 tape. Both the TQK50 and the TUK50 are TMSCP devices.

Section 6.1.12 describes compatibility among the TK50, TK70, and TZ30 magnetic tape cartridge systems.

6.1.10 TK70 Cartridge Tape System

The TK70 is a 295-MB, 5.25-inch, streaming cartridge tape system. (See Section 6.2.7 for information about streaming tape technology.) The TK70 tape drive records data serially on 48 tracks using serpentine recording, rather than separate (parallel) tracks. Data written to the tape is automatically read as it is written. A CRC check is performed and the controller is notified immediately if an error occurs on the tape.

The TQK70 is a dual-height, Q-bus controller for the TK70 magnetic tape drive. The TK70 subsystem includes a 38-KB cache to optimize performance. The TBK70 is a VAXBI-bus controller for the same drive. Section 6.1.12 describes compatibility between the TK50 and TK70 magnetic tape cartridge systems.

6.1.11 TZ30 Cartridge Tape System

The TZ30 is a 95-MB, 5.25-inch, half-height cartridge streaming tape drive with an embedded SCSI controller. See Section 6.2.7 for information about streaming tape technology. The TZ30 uses TK50 cartridge tapes. It records data serially on 22 tracks using serpentine recording. Section 6.1.12 describes compatibility between the TK50, TK70, and TZ30 magnetic tape cartridge systems.

6.1.12 Read and Write Compatibility Among Cartridge Tape Systems

When you insert a cartridge tape into the TZ30, TK50, and TK70 tape drives, the hardware initializes the media to a device-specific recording density automatically.

Depending on the type of cartridge and the type of drive on which it is formatted (inserted and initialized), full read and write access to tape cartridges may not be permitted.

Formatting a Blank TK50 Cartridge Tape

A blank, unformatted TK50 cartridge can be formatted on the TK50, TK70, and TZ30 cartridge systems. For example, a TK70 tape drive has full read and write access to a TK50 cartridge formatted on a TK70 drive. Once the cartridge tape is formatted on a particular tape drive, the tape drive has full read and write access to the cartridge tape.

Formatting a Previously Initialized TK50 Cartridge Tape

If a TK50 cartridge tape is formatted on a TZ30 or TK50 cartridge tape drive, the TZ30 and TK50 drives initialize the TK50 cartridge to TK50 density. The following table summarizes the types of access available:

	TK50	
Controller	**Read**	**Write**
TZ30[1]	Yes	Yes
TQK50	Yes	Yes
TQK70	Yes	No

[1]Has an internal controller.

The TK70 tape drive can read data on a TK50 cartridge formatted on a TK50 or TZ30 tape drive.

Formatting a TK50 Cartridge Tape on a TK70 Tape Drive

If a TK50 or TK52 cartridge tape is formatted on a TK70 tape drive, the TK70 cartridge tape drive initializes the TK50 or TK52 cartridge tape to TK70 density. The following table summarizes the types of access available:

	TK50		TK52	
Controller	**Read**	**Write**	**Read**	**Write**
TZ30[1]	No	No	No	No
TQK50	No	No	No	No
TQK70	Yes	Yes	Yes	Yes

[1]Has an internal controller.

The TK50 and TZ30 tape drives cannot read or write data on a TK50 cartridge tape formatted on a TK70 drive.

6.2 Driver Features

The VMS magnetic tape drivers provide the following features:

- Multiple master adapters and slave formatters
- Different types of devices on a single MASSBUS adapter; for example, an RP05 disk and a TM03 tape formatter
- Reverse read function (except for the TZ30 and TK50 on TUK50 and TQK50 controllers)
- Reverse data check function (except for TZ30, TS11, and TK50 on TUK50 and TQK50 controllers)
- Data checks on a per-request, per-file, or per-volume basis (except for TS11)

- Full recovery from power failure for online drives with volumes mounted, including repositioning by the driver (except on VAXstation 2000 and MicroVAX 2000 systems)
- 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.

6.2.1 Dual Path Tape Drives

A **dual-path** HSC tape drive is a drive that connects to two HSCs, both of which have the same nonzero tape allocation class. The VMS operating system recognizes the dual-pathed capability of such a tape drive under the following circumstances: (1) the VMS operating system has access to both HSCs and (2) select buttons for both ports are depressed on the tape drive.

If one port fails, the VMS operating system switches access to the operational port automatically, provided that the allocation class information has been defined correctly.

6.2.2 Dynamic Failover and Mount Verification

Dynamic failover occurs on dual-pathed tape drives if mount verification is unable to recover on the current path and an alternate path is available. The failover occurs automatically and transparently and then mount verification proceeds.

A device enters mount verification when I/O request fails because the device has become inoperative. This might occur in the following instances:

- The device is place offline accidentally.
- The active port of an HSC-connected drive fails.
- A hardware error occurs.
- The device is set to write protected during a write operation.

When the device comes back online, either through automatic failover or operator intervention, the VMS operating system validates the volume, restores the tape to the position when the I/O failure occurred, and retries the failed request.

6.2.3 Tape Caching

The RV20, TA90, TK70, and TU81-Plus contain **write-back volatile** caches. The host enables write-back volatile caches explicitly, either on a per-unit basis or on a per-command basis. To enable caching on a per-unit basis, the user can enter the DCL MOUNT command specifying the qualifier /CACHE=TAPE_DATA.

The VMS Backup Utility enables caching on a per-command basis. The user can implement caching on a per-command basis at the QIO level by using the IO$M_NOWAIT function modifiers on commands where it is legal. (See Table 6–5.) In the unlikely event that cached data is lost, the system returns a fatal error and the device accepts no further I/O requests. The IO$M_FLUSH function code can be used to ensure that all write-back-cached data has been written out to the specified tape unit. The IO$_PACKACK, IO$_UNLOAD, IO$_REWINDOFF, and IO$_AVAILABLE function codes also flush the cache.

6.2.4 Master Adapters and Slave Formatters

The VMS operating system supports the use of many 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.

The VMS operating system also supports the use of multiple slave formatters per master adapter on a system. For example, more than one TM03 or TM78 magnetic tape formatter per MBA can be used on a system. A slave formatter accepts data and commands from a master adapter and directs the operation of one or more slave drives. The TM03 and the TM78 are slave formatters. The TE16, TU45, TU77, and TU78 magnetic tape drives are slave drives.

6.2.5 Data Check

After successful completion of an I/O operation, a data check is made to compare the data in memory with that on the tape. After a write or read (forward) operation, the tape drive spaces backward, and then performs a write check data operation. After a read operation in the reverse direction, the tape drive spaces forward, and then performs a write check data reverse operation. With the exception of TS04 and TU80 drives, magnetic tape drivers support data checks at the following three levels:

- Per request—You 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—You can specify the characteristics "data check all reads" and "data check all writes" when the volume is mounted. The *VMS DCL Dictionary* describes volume mounting and dismounting. The

VMS System Services Reference Manual describes the Mount Volume ($MOUNT) and Dismount Volume ($DISMOU) system services.

- Per file—You 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 1 of this manual and the *VMS Record Management Services Manual* both describe file access.

Data check is distinguished from a BACKUP/VERIFY operation, which writes an entire save set, rewinds, and then compares the tape to the original tape.

See Section 6.1.9 for information on TK50 data check.

Note: Read and write operations with data check can result in very slow performance on streaming tape drives.

6.2.6 Error Recovery

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

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

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

Power failure recovery consists of repositioning the reel to the position held at the start of the I/O operation in progress at the time of the power failure, and then reexecuting this operation. This repositioning might or might not require operator intervention to reload the drives. When such operator intervention is required, "device not ready" messages are sent to the operator console to solicit reloading of mounted drives. Power failure recovery is not supported on VAXstation 2000 and MicroVAX 2000 systems.

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.

If a nonfatal controller/drive error occurs, the driver (or the controller, depending on the type of drive) attempts to reexecute the I/O operation up to 16 times before returning a fatal error. The driver repositions the tape before each retry.

The inhibit retry function modifier (IO$M_INHRETRY) inhibits all normal (nonspecial conditions) error recovery. If an error occurs, and the request includes that modifier, the operation is immediately terminated and the driver returns a failure status. IO$M_INHRETRY has no effect on power failure and timeout recovery.

The driver can write up to 16 extended interrecord gaps during the error recovery for a write operation. For the TE16, TU45, and TU77, writing these gaps can be suppressed by specifying the inhibit extended interrecord gap function modifier (IO$M_INHEXTGAP). This modifier is ignored for the other magnetic tape drives.

6.2.7 Streaming Tape Systems

Streaming tape systems, such as the TK50, TK70,TU80, TU81, TU81-Plus, TA81, and TZ30, use the supply and takeup reel mechanisms to control tape speed and tension directly, thereby eliminating the need for more complex and costly tension and drive components. Streaming tapes have a very simple tape path, much like a home audio reel-to-reel recorder.

Note: Read and write operations with data check can result in very slow performance on streaming tape drives.

Because the motors driving the reels are low-powered and because there is no tape buffering, streaming tape drives are not capable of starting and stopping in the interrecord gaps like conventional tape drives. When a streaming tape does have to stop, the following events occur:

1. The tape slowly coasts forward to a stop.
2. It backs up over a section previously processed.
3. It halts to await the next command.
4. It accelerates so that, when the original interrecord gap is encountered, the tape is moving at full speed.

These steps, allowing the tape to reposition, require approximately one-half second to complete on TU8x tapes and about three seconds on TK50 tapes. If the operating system is not capable of writing to, or reading from, a streaming tape drive at a rate that will keep the drive in constant motion (streaming) the drive repositions itself when it runs out of commands to execute. That produces a situation known as *thrashing*, in which the relatively long reposition times exceed the time spent processing data and the result is lower-than-expected data throughput.

Thrashing is entirely dependent on how fast the system can process data relative to the tape drive speed while streaming. Consequently, the greatest efficiency is obtained when you provide sufficient buffering to ensure continuous tape motion. Some streaming tape drives supported by the VMS operating system (TU80, TU81, TU81-Plus, and TA81) are dual-speed devices that automatically adjust the tape speed to maximize data throughput and minimize thrashing.

The TK50 writes up to seven filler records to keep the tape in motion. These records are ignored when the data is read.

6.3 Magnetic Tape Driver Device Information

You can obtain information on all magnetic tape device characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *VMS System Services Reference Manual*.)

$GETDVI returns magnetic tape characteristics when you specify the item codes DVI$_DEVCHAR, DVI$_DEVCHAR2, DVI$_DEVDEPEND, and DVI$_DEVDEPEND2. Tables 6–2, 6–3, and 6–4 list these characteristics. The $DEVDEF macro defines the device-independent characteristics, the $MTDEF macro defines the device-dependent characteristics, and the $MT2DEF macro defines the extended device characteristics. The extended device characteristics apply only to the TU81-Plus.

Table 6–2 Magnetic Tape Device-Independent Characteristics

Characteristic[1]	Meaning
Dynamic Bits (Conditionally Set)	
DEV$M_AVL	Device is online and available.
DEV$M_FOR	Volume is foreign.
DEV$M_MNT	Volume is mounted.
DEV$M_RCK	Perform data check on all read operations.
DEV$M_WCK	Perform data check on all write operations.
Static Bits (Always Set)	
DEV$M_FOD	Device is file-oriented.
DEV$M_IDV	Device is capable of input.
DEV$M_ODV	Device is capable of output.
DEV$M_SQD	Device is capable of sequential access.
DEV$M_WBC[2]	Device is capable of write-back caching.

[1] Defined by the $DEVDEF macro.

[2] This bit is located in DVI$_DEVCHAR2.

Table 6–3 Device-Dependent Information for Tape Devices

Characteristic[1]	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 the tape in the forward direction.

[1] Defined by the $MTDEF macro.

(continued on next page)

Table 6–3 (Cont.) Device-Dependent Information for Tape Devices

Characteristic[1]	Meaning
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$M_PARITY	If set, all data transfers are performed with even parity. If clear (normal case), all data transfers are performed with odd parity. Only non-return-to-zero-inverted recording at 800 bpi can have even parity.
MT$V_DENSITY MT$S_DENSITY	Specifies the density at which all data transfer operations are performed. Possible density values are as follows: MT$K_GCR_6250 Group-coded recording, 6250 bpi MT$K_PE_1600 Phase-encoded recording, 1600 bpi MT$K_NRZI_800 Non-return-to-zero-inverted recording, 800 bpi MT$K_BLK_833 Cartridge block mode recording[2]
MT$V_FORMAT MT$S_FORMAT	Specifies the format in which all data transfers are performed. A possible format value is as follows: MT$K_NORMAL11 Normal PDP–11 format. Data bytes are recorded sequentially on tape with each byte occupying exactly one frame.

[1]Defined by the $MTDEF macro.

[2]Only for the TK50 and TZ30.

Table 6–4 Extended Device Characteristics for Tape Devices

Characteristic[1]	Meaning
MT2$V_WBC_ENABLE	If set, write-back caching is enabled for this unit.
MT2$V_RDC_DISABLE	If set, read caching is disabled for this unit.

[1]Defined by the $MT2DEF macro. Only for the TU81-Plus. Initial device status will show both of these bits cleared; write-back caching will be disabled, read caching will be enabled.

DVI$_DEVTYPE and DVI$_DEVCLASS return the device type and class names, which are defined by the $DCDEF macro. DVI$_DEVBUFSIZ returns the buffer size. The buffer size is the default to be used for tape transfers (normally 2048 bytes). The device class for magnetic tapes is $DCTAPE, and the device type is determined by the magnetic tape model. For example, the device type for the TA78 is DT$_TA78, for the TA81 it is DT$_TA81.

6.4 Magnetic Tape Function Codes

The VMS magnetic tape driver can perform logical, virtual, and physical I/O functions. Foreign-mounted devices do not require privilege to perform logical and virtual I/O requests.

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. The results of logical and physical I/O operations are unpredictable if an ACP is present.

Virtual I/O functions require intervention by an ACP and must be executed in a prescribed order. The normal order is to create and access a file, write information to that file, and deaccess the file. Subsequently, when you access the file, you read the information and then deaccess the file. You can write over the file when the information it contains is no longer useful and the file has expired.

Any number of bytes (from a minimum of 14 to a maximum of 65,535) 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 for the function actually to be executed. If it is not, either a "device not mounted" or "invalid volume" status is returned in the I/O status block.

Table 6–5 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 1 describes the QIO level interface to the magnetic tape device ACP.

Table 6–5 Magnetic Tape I/O Functions

Function Code	Arguments	Type[1]	Function Modifiers	Function
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$M_CREATE is set, create a file at end-of-tape (EOT) marker.
IO$_ACPCONTROL	P1,[P2],[P3],[P4], [P5]	V	IO$M_DMOUNT	Perform miscellaneous control functions.[9]
IO$_AVAILABLE		P		Clear volume valid bit.
IO$_CREATE	P1,[P2[,[P3],[P4],[P5]	V	IO$M_CREATE IO$M_ACCESS	Create a file.

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

[9] See Section 1.6.7 for additional information.

(continued on next page)

Table 6–5 (Cont.) Magnetic Tape I/O Functions

Function Code	Arguments	Type[1]	Function Modifiers	Function
IO$_DEACCESS	P1,[P2],[P3],[P4],[P5]	V		Deaccess a file and, if the file has been written, write out trailer records.
IO$_DSE[2]		P	IO$M_NOWAIT	Erase a prescribed section of the tape.
IO$_FLUSH		L		Flush the controller cache to tape.
IO$_MODIFY	P1,[P2],[P3],[P4],[P5]	V		Write user labels.
IO$_PACKACK		P		Initialize volume valid bit.
IO$_READLBLK	P1,P2	L	IO$M_DATACHECK[3] IO$M_INHRETRY IO$M_REVERSE[4]	Read logical block.
IO$_READPBLK	P1,P2	P	IO$M_DATACHECK[3] IO$M_INHRETRY IO$M_REVERSE[4]	Read physical block.
IO$_READVBLK	P1,P2	V	IO$M_DATACHECK[3] IO$M_INHRETRY IO$M_REVERSE[4]	Read virtual block.
IO$_REWIND		L	IO$M_INHRETRY IO$M_NOWAIT	Reposition tape to the beginning-of-tape (BOT) marker.
IO$_REWINDOFF		L	IO$M_INHRETRY IO$M_NOWAIT	Rewind and unload the tape on the selected drive.
IO$_SENSECHAR	[P1] [P2][8]	P	IO$M_INHRETRY	Sense the tape characteristics and return them in the I/O status block.
IO$_SENSEMODE	[P1][P2][8]	L	IO$M_INHRETRY	Sense the tape characteristics and return them in the I/O status block.
IO$_SETCHAR	P1,[P2][8]	P		Set tape characteristics for subsequent operations.
IO$_SETMODE	P1,[P2][8]	L		Set tape characteristics for subsequent operations.
IO$_SKIPFILE	P1	L	IO$M_INHRETRY IO$M_NOWAIT[5]	Skip past a specified number of tape marks in either a forward or reverse direction.

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

[2]Only for TMSCP drives, TZK50, and TZ30.

[3]Not for TS04 and TU80.

[4]Not for TUK50 and TQK50.

[5]Only for RV20, TA90, TK70, and TU81-Plus drives.

[8]The P1 and P2 arguments for IO$_SENSEMODE and IO$_SENSECHAR and the P2 argument for IO$_SETMODE and IO$_SETCHAR are for TMSCP drives only.

(continued on next page)

Table 6–5 (Cont.) Magnetic Tape I/O Functions

Function Code	Arguments	Type[1]	Function Modifiers	Function
IO$_SKIPRECORD	P1	L	IO$M_INHRETRY IO$M_NOWAIT[5]	Skip past a specified number of blocks in either a forward or reverse direction.
IO$_UNLOAD		L	IO$M_INHRETRY IO$M_NOWAIT	Rewind and unload the tape on the selected drive.
IO$_WRITELBLK	P1,P2	L	IO$M_ERASE[6] IO$M_DATACHECK[3] IO$M_INHRETRY IO$M_INHEXTGAP[7] IO$M_NOWAIT[5]	Write logical block.
IO$_WRITEOF		L	IO$M_INHRETRY IO$M_INHEXTGAP[7] IO$M_NOWAIT[5]	Write an extended interrecord gap followed by a tape mark.
IO$_WRITEPBLK	P1,P2	P	IO$M_ERASE[6] IO$M_DATACHECK[3] IO$M_INHRETRY IO$M_INHEXTGAP[7] IO$M_NOWAIT[5]	Write physical block.
IO$_WRITEVBLK	P1,P2	V	IO$M_DATACHECK[3] IO$M_INHRETRY IO$M_INHEXTGAP[7] IO$M_NOWAIT[5]	Write virtual block.

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

[3]Not for TS04 and TU80.

[5]Only for RV20, TA90, TK70, and TU81-Plus drives.

[6]Takes no arguments; valid only for TMSCP drives, TZK50, and TZ30.

[7]Only for TE16, TU45, and TU77.

The function-dependent arguments for IO$_CREATE, IO$_ACCESS, IO$_DEACCESS, IO$_MODIFY, IO$_ACPCONTROL are as follows:

- P1—The address of the file information block (FIB) descriptor.
- P2—Optional. The address of the file name string descriptor. 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—Optional. The address of the word that is to receive the length of the resultant file name string.
- P4—Optional. The address of a descriptor for a buffer that is to receive the resultant file name string.
- P5—Optional. The address of a list of attribute descriptors. 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 1 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 as follows:

- 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 length of the buffer specified by P1

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.)

The following example shows the correct method of defining the P1 parameter in a IO$_SKIPRECORD QIO.

```
        .
        .
        .
TAPE_CHAN:
        .WORD    0
IOSB:   .WORD    0
        .WORD    0
        .LONG    0
DEVICE: .ASCID   /$127$MUA0:/
RECORD: .LONG    2000
;
        .PSECT   CODE,EXE,NOWRT
;
        .ENTRY   MT_IO,^M<>
;
        $ASSIGN_S    CHAN=TAPE_CHAN,-
                     DEVNAM=DEVICE
        BLBC     R0,EXIT_ERROR
;
        $QIOW_S  CHAN=TAPE_CHAN,-
                 FUNC=#IO$_SKIPRECORD,-
                 IOSB=IOSB,-
                 P1=RECORD
        BLBC     R0,EXIT_ERROR
        $EXIT_S  R0
        .
        .
        .
EXIT_ERROR:
        $EXIT_S  R0
        .END     MT_IO
```

6.4.1 Read

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

The VMS operating system provides the following 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, it is converted to a read logical block function. If a read virtual block function is directed to a volume that is mounted structured, the volume is handled the same way as a file-structured device.

Two function-dependent arguments are used with these codes: P1 and P2. These arguments are described in Section 6.4.

If the read function code includes the reverse function modifier (IO$M_REVERSE), the drive reads the tape in the reverse direction instead of the forward direction. IO$M_REVERSE cannot be specified for the TUK50 and TQK50 devices.

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 completes. (The drive performs a space reverse or space forward between the read and data check operations.) A data check operation is also performed if the volume that was 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 TS04 and TU80 tape drives do not support the data check function.

For read physical block and read logical block functions, the drive returns the status SS$_NORMAL (not end-of-tape status) if either of the following conditions occurs and no other error condition exists:

- The tape is positioned past the end-of-tape (EOT) position at the start of the read (forward or reverse) operation.
- The tape enters the EOT region as a result of the read (forward) operation.

The transferred byte count reflects the actual number of bytes read.

If the drive reads a tape mark during a logical or physical read operation in either the forward or reverse direction, any of the following conditions can return an end-of-file status:

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

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

If the drive attempts to read a block that is larger than the specified memory buffer during a logical or physical read operation, a data overrun status is returned. The buffer receives only the first part of the block. On a read in the reverse direction (on drives other than the TK50 and TZ30) the buffer receives only the latter part of the block. 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. The TUK50 and TZ30 cannot actually perform read reverse operations; they must be simulated by the driver. Therefore, the data returned are those that would have been returned had the block been read in the forward direction.

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

6.4.2 Write

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

The VMS operating system provides the following 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 a write logical block. If a write virtual block function is directed to a volume that is mounted structured, the volume is handled the same way as a file-structured device.

Two function-dependent arguments are used with these codes: P1 and P2. These arguments are described in Section 6.4.

The IO$M_ERASE function modifier can be used with the IO$_WRITELBLK and IO$_WRITEPBLK function codes to erase a user-selected part of a tape. This modifier propagates an erase pattern of all zeros from the current tape position to 10 feet past the EOT position and then rewinds to the BOT marker.

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 completes. (The drive performs a space reverse between the write and the data check operations.) The driver forces a data check operation when an error occurs during a write operation. This ensures that the data can be reread. 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 TS04 and TU80 tape drives do not support the data check function.

If the IO$M_NOWAIT function modifier is specified, write-back caching is enabled on a per command basis. IO$M_NOWAIT is applicable only to TU81-Plus drives.

If the drive performs a write physical block or a write logical block operation, an EOT status is returned if either of the following conditions occurs and no other error condition exists:

- The tape is positioned past the EOT position at the start of the write operation.
- The tape enters the EOT 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.

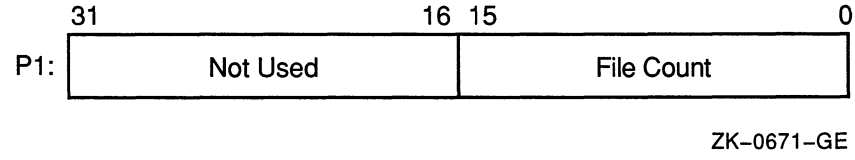
6.4.3 Rewind

The rewind 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.

6.4.4 Skip File

The skip file 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 6–1. 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 as an unsigned number in the I/O status block.)

Figure 6–1 IO$_SKIPFILE Argument



ZK–0671–GE

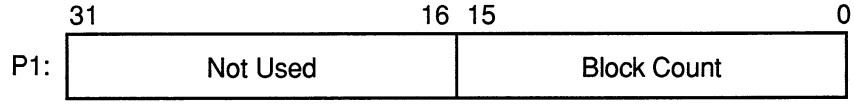
Only tape marks (when the tape moves in either direction) and the BOT marker (when the tape moves in reverse) are counted during a skip file operation. The BOT marker terminates a skip file function in the reverse direction. The end-of-tape (EOT) marker does not terminate a skip file function in either the forward or reverse direction. A negative skip file function leaves the tape positioned just before a tape mark (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.

A skip file function in the forward direction can also be terminated if two consecutive tape marks are encountered. Section 6.4.5.1 describes this feature.

6.4.5 Skip Record

The skip record function skips past a specified number of physical tape blocks in either a forward or reverse direction. A device- or function-dependent argument (P1) specifies the number of blocks to skip, as shown in Figure 6–2. 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 as an unsigned number in the I/O status block. If a tape mark is detected, the count is the number of blocks skipped, plus 1 (forward tape motion) or minus 1 (reverse tape motion).

Figure 6–2 IO$_SKIPRECORD Argument



ZK–0672–GE

A skip record operation is terminated by the end-of-file marker 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.

A skip record function in the forward direction can also be terminated if the tape was originally positioned between two tape marks. Section 6.4.5.1 describes this feature.

6.4.5.1 Logical End-of-Volume Detection

A skip file or skip record operation is terminated when two consecutive tape marks are encountered when the tape moves in the forward direction.

After the operation terminates, the tape remains positioned between the two tape marks that were detected. The I/O status block (IOSB) returns the status SS$_ENDOFVOLUME and the actual number of files (or records) skipped during the operation prior to the detection of the second tape mark. The skip count is returned in the high-order word of the first longword of the IOSB.

Subsequent skip record (or skip file) requests terminate immediately when the tape is positioned between the two tape marks, producing no net tape movement and returning the SS$_ENDOFVOLUME status with a skip count of zero.

To move the tape beyond the second tape mark, you must employ another I/O function. For example, the IO$_READLBLK function, if issued after receipt of the SS$_ENDOFVOLUME status return, terminates with an SS$_ENDOFFILE status and with the tape positioned just past the second tape mark. From this new position, other skip functions could be issued to produce forward tape motion (assuming there is additional data on the tape).

If three consecutive tape marks are encountered during a skip file function, you must issue two IO$_READLBLK functions, the first to get the SS$_ENDOFFILE return, the second to position the tape past the third tape mark.

6.4.6 Write End-of-File

The write-end-of-file function writes an extended interrecord gap (of approximately 3 inches for non-return-to-zero-inverted (NRZI) recording and 1.5 inches for phase-encoded (PE) recording) followed by a tape mark. No device- or function-dependent arguments are used with IO$_WRITEOF.

An end-of-tape (EOT) 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 EOT region.

6.4.7 Rewind Offline

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

6.4.8 Unload

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

6.4.9 Sense Tape Mode

The sense tape mode function senses the current device-dependent and extended device characteristics (see Tables 6–3 and 6–4).

The VMS operating system provides the following function codes:

- IO$_SENSEMODE—Sense mode
- IO$_SENSECHAR—Sense characteristics

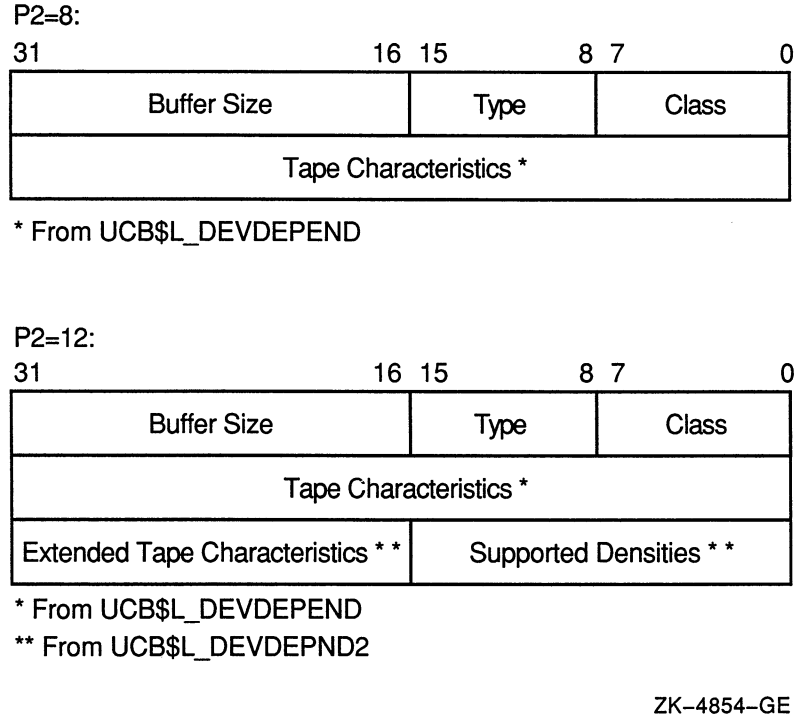
Sense mode requires logical I/O privilege. Sense characteristics requires physical I/O privilege. For TMSCP drives the sense mode function returns magnetic tape information in a user-supplied buffer, which is specified by the following function-dependent arguments:

- P1—Optional. Address of a user-supplied buffer.
- P2—Optional. Length of a user-supplied buffer.

If P1 is not zero, the sense mode buffer returns the tape characteristics. (If P2=8, the second longword of the buffer contains the device-dependent characteristics. If P2=12, the second longword contains the device-dependent characteristics and the third longword contains the tape densities that the drive supports and the extended tape characteristics.) The extended characteristics are identical to the information returned by DVI$_DEVDEPEND2 (see Table 6–4). Figure 6–3 shows the contents of the P1 buffer.

Regardless of whether the P1 buffer is specified, the I/O status block returns the device-dependent characteristics in the second longword (see Figure 6–6). These characteristics are identical to the information returned by DVI$_DEVDEPEND (see Table 6–3 in Section 6.3).

Figure 6–3 Sense Mode P1 Buffer



6.4.10 Set Mode

Set mode operations affect the operation and characteristics of the associated magnetic tape device. The VMS operating system defines 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. The following function codes are provided:

- IO$_SETMODE—Set mode
- IO$_SETCHAR—Set characteristics

These functions take the following device- or function-dependent arguments (other arguments are ignored):

- P1—The address of a characteristics buffer
- P2—Optional. The length of the characteristics buffer. Default is eight bytes. If a length of 12 bytes is specified, the third longword (which is for TMSCP drives only) specifies the extended tape characteristics.

Figure 6–4 shows the P1 characteristics buffer for IO$_SETMODE. Figure 6–5 shows the same buffer for IO$_SETCHAR.

Figure 6–4 Set Mode Characteristics Buffer

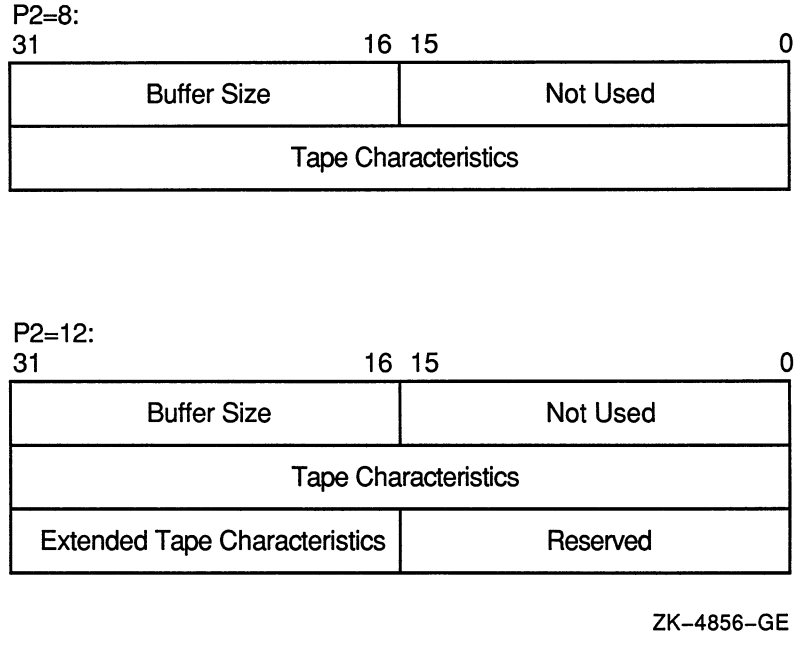
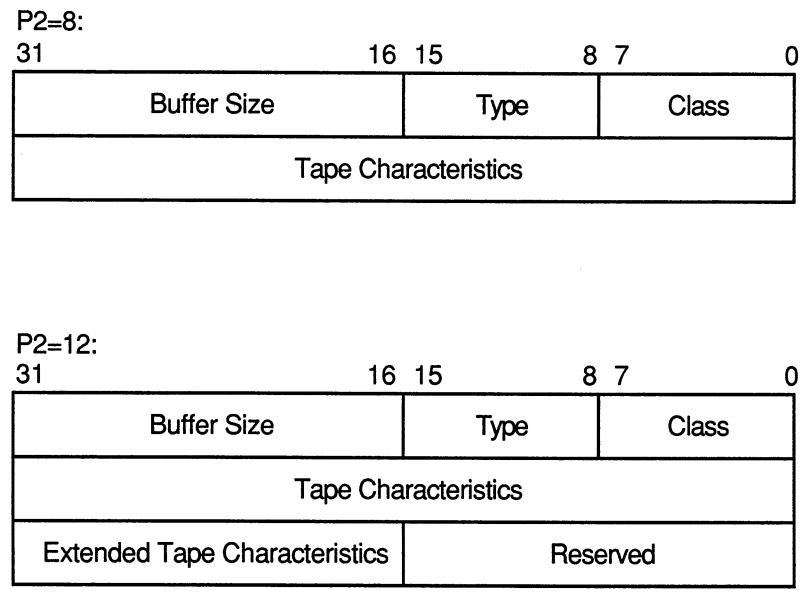


Figure 6–5 Set Characteristics Buffer



The first longword of the P1 buffer for the set characteristics function contains information on device class and type, and the buffer size. The device class for tapes is DC$_TAPE.

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 second longword of the P1 buffer for both the set mode and set characteristics functions contains the tape characteristics. Table 6–6 lists the tape characteristics and their meanings. The $MTDEF macro defines the symbols listed. If P2=12, the third longword contains the extended tape characteristics for TMSCP drives, which are listed in Table 6–7. The extended tape characteristics are defined by the $MT2DEF macro and are identical to the information returned by DVI$_DEVDEPEND2.

Table 6–6 Set Mode and Set Characteristics Magnetic Tape Characteristics

Characteristic[1]	Meaning
MT$M_PARITY	If set, all data transfers are performed with even parity. If clear (normal case), all data transfers are performed with odd parity. Even parity can be selected only for non-return-to-zero-inverted recording at 800 bpi. Even parity cannot be selected for phase-encoded recording (tape density is MT$K_PE_1600) or group-coded recording (tape density is MT$K_GCR_6250) and is ignored.
MT$V_DENSITY MT$S_DENSITY	Specifies the density at which all data transfers are performed. Tape density can be set only when the selected drive's tape position is at the BOT marker. Possible density values are as follows: MT$K_DEFAULT — Default system density. MT$K_GCR_6250 — Group-coded recording, 6250 bpi. MT$K_PE_1600 — Phase-encoded recording, 1600 bpi. MT$K_NRZI_800 — Non-return-to-zero-inverted recording, 800 bpi. MT$K_BLK_833 — Cartridge block mode recording[2].
MT$V_FORMAT MT$S_FORMAT	Specifies the format in which all data transfers are performed. Possible format values are as follows: 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.

[1] Defined by the $MTDEF macro

[2] Only for the TK50 and TZ30

Table 6–7 Extended Device Characteristics for Tape Devices

Characteristic[1]	Meaning
MT2$V_WBC_ENABLE	Enable write-back caching on a per unit basis.
MT2$V_RDC_DISABLE	Disable read caching on a per unit basis.

[1] Defined by the $MT2DEF macro. Only for TU81-Plus drives.

Application programs that change specific magnetic tape characteristics should perform the following steps, as shown in Example 6–2 in Section 6.6:

1 Use the IO$_SENSEMODE function to read the current characteristics.

2 Modify the characteristics.

3 Use the set mode function to write back the results.

Failure to follow this sequence will result in clearing any previously set characteristic.

6.4.11 Data Security Erase

The data security erase function erases all data from the current position of the volume to 10 feet beyond the EOT reflective strip and then rewinds the tape to the BOT marker. It is a physical I/O function and requires the access privilege necessary to perform physical I/O functions. It is applicable only for the TA78, TU78, TA81, TK50, TU81, TU81-Plus, and TZ30 drives. The following function code is provided:

- IO$_DSE

If the function is issued when a tape is positioned at the BOT marker, all data on the tape will be erased.

IO$_DSE takes no device- or function-dependent arguments.

6.4.12 Pack Acknowledge

The pack acknowledge function sets the volume valid bit for all magnetic tape devices. It is a physical I/O function and requires the access privilege to perform physical I/O. The following function code is provided:

- IO$_PACKACK

This function code takes no function-dependent arguments.

IO$_PACKACK must be the first function issued when a volume is placed in a magnetic tape drive. IO$_PACKACK is issued automatically when the DCL commands INITIALIZE or MOUNT are issued.

6.4.13 Available

The available function clears the volume valid bit for all magnetic tape drives, that is, it reverses the function performed by the pack acknowledge function (see Section 6.4.12). A rewind of the tape is performed (applicable to all tape drives). No unload function is issued to the drive. The following function code is provided:

- IO$_AVAILABLE

This function takes no function-dependent arguments.

6.4.14 Flush

The flush function is used to ensure that all previously issued cached commands have fully completed. Normally, hosts use this function to establish or maintain synchronization with write-back cached commands issued to the specified tape unit. The I/O request does not complete until all cached data is written successfully to the media in the exact order that the user specified.

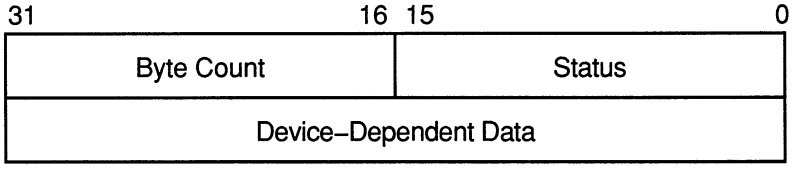
- IO$_FLUSH

This function code takes no function-dependent arguments.

6.5 I/O Status Block

The I/O status block (IOSB) for QIO functions on magnetic tape devices is shown in Figure 6–6. Appendix A lists the status returns for these functions. (The *VMS System Messages and Recovery Procedures Reference Manual* provides explanations and suggested user actions for these returns.) Table 6–3 (in Section 6.3) lists the device-dependent data returned in the second longword. The IO$_SENSEMODE function can be used to return that data.

Figure 6–6 IOSB Contents



ZK–0675–GE

The byte count is the actual number of bytes transferred to or from the process buffer or the number of files or blocks skipped. (If a IO$_SKIPRECORD function is terminated by the detection of a tape mark, the count returned in the IOSB is an unsigned number reflecting the number of blocks skipped, plus 1.

6.6 Magnetic Tape Driver Programming Examples

This section presents three magnetic tape driver programming examples.

6.6.1 Magnetic Tape Data Program Example

Example 6–1 shows 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 a magnetic tape driver. However, this would have involved additional programming such as writing header labels and trailer labels.

Example 6–1 Magnetic Tape Data Program Example

```
; ****************************************************************
;
        .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                     ;Initialize buffer to
        .ASCII  /A/                     ;contain 'A'
        .ENDR                           ;
;
; Now define the file information block (FIB), which the ACP uses
; in accessing and deaccessing the file.  Both the user and the ACP
; supply the information required in the FIB to perform these
; functions.
;
```

(continued on next page)

Example 6–1 (Cont.) Magnetic Tape Data Program Example

```
FIB_DESCR:                                  ;Start of FIB
        .LONG   ENDFIB-FIB                  ;Length of FIB
        .LONG   FIB                         ;Address of FIB
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

;
; 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
;
; ****************************************************************
;
;                       Start Program
;
; ****************************************************************
;
; The program first assigns a channel to the magnetic tape unit and
; then performs an access function to create and access a file called
; MYDATA.DAT.  Next, the program writes 26 blocks of data (the letters
; of the alphabet) to the tape.  The first block contains all A's, the
; next, all B's, and so forth.  The program starts by writing a block of
; 256 bytes, that is, the block of A's.  Each subsequent block is reduced
; in size by two bytes so that by the time the block of Z's is written,
; the size is only 206 bytes.  The magtape ACP does not allow the reading
; of a file that has been written until one of three events occurs:
;       1. The file is deaccessed.
;       2. The file is rewound.
;       3. The file is backspaced.
; In this example the file is backspaced zero blocks and then read in
; reverse (incrementing the block size every block); the data is
; checked against the data that is supposed to be there.  If no data
; errors are detected, the file is deaccessed and the program exits.
;
        .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          ;Success?
        BSBW    ERRCHECK                ;Find out

;
; Now create and access the file MYDATA.DAT.
;
```

(continued on next page)

Example 6–1 (Cont.) Magnetic Tape Data Program Example

```
        $QIOW_S CHAN=TAPECHAN,-          ;Channel is magtape
                FUNC=#IO$_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           ;Success?
        BSBW    ERRCHECK                 ;Find out

;
; LOOP1 consists of writing the alphabet to the tape (see previous
; description).
;
        MOVL    #26,R5                   ;Set up loop count
        MOVL    #256,R3                  ;Set up initial byte count
                                         ;in R3
LOOP1:                                   ;Start of loop
        $QIOW_S CHAN=TAPECHAN,-          ;Perform QIOW to tape channel
                FUNC=#IO$_WRITEVBLK,-    ;Function is write virtual
                -                        ;block
                P1=BUFFER,-              ;Buffer address
                P2=R3                    ;Byte count
        CMPW    #SS$_NORMAL,R0           ;Success?
        BSBW    ERRCHECK                 ;Find out
;
; Now decrement the byte count in preparation for the next write
; operation and set up a loop count for updating the character
; written; 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

;
; The alphabet is now complete.  Fall through LOOP1 and update the
; byte count so that it reflects the actual size of the last block
; written to tape.
;
        ADDL2   #2,R3                    ;Update byte count

;
; The tape is now read, but first the program must perform one of
; the three functions described previously before the ACP allows
; read access.  The program performs an ACP control function,
; specifying skip zero blocks.  This is a special case of skip reverse
; and causes the ACP to allow read access.
;
```

(continued on next page)

Example 6–1 (Cont.) Magnetic Tape Data Program Example

```
        CLRL    FIB+FIB$L_CNTRLVAL      ;Set up to space zero blocks
        MOVW    #FIB$C_SPACE,FIB+FIB$W_CNTRLFUNC ;Set up for space
                                        ;function
        $QIOW_S CHAN=TAPECHAN,-         ;Perform QIOW to tape channel
                FUNC=#IO$_ACPCONTROL,-  ;Perform an ACP control
                -                       ;function
                P1=FIB_DESCR            ;Define the FIB
        CMPW    #SS$_NORMAL,R0          ;Success?
        BSBW    ERRCHECK                ;Find out

;
; 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

;
; Check the data read to verify that it matches the data written.
;
        MOVL    R3,R4                   ;Copy R3 to R4 for loop count
CHECKDATA:                              ;
        CMPB    (R7)+,R6                ;Check each character
        BNEQ    MISMATCH                ;If error, print message
        SOBGTR  R4,CHECKDATA            ;Continue until finished
        DECB    R6                      ;Go through alphabet in reverse
        ADDL2   #2,R3                   ;Update byte count by 2 for
                                        ;next block
        SOBGTR  R5,LOOP3                ;Read next block

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

;
; Deassign the channel and exit.
;
EXIT:   $DASSGN_S CHAN=TAPECHAN         ;Deassign channel
        RET                             ;Exit

;
; If an error had been detected, a program would normally
; generate an error message here.  But for this example the
; program simply exits.
;
```

(continued on next page)

Example 6–1 (Cont.) Magnetic Tape Data Program Example

```
MISMATCH:                                   ;
        BRB     EXIT                        ;Exit

ERRCHECK:                                   ;
        BNEQ    EXIT                        ;If not success, exit
        RSB                                 ;Otherwise, return

        .END    MAGTAPE_EXAMPLE
```

6.6.2 Magnetic Tape Device Characteristic Program Example

Example 6–2 illustrates the recommended sequence for changing a device characteristic. Retrieve the current characteristics using a IO$_SENSEMODE request, set the new characteristics bits, and then use IO$_SETMODE to set the new characteristics.

Example 6–2 Device Characteristic Program Example

```
$QIOW_S -                                   ; Get current characteristics.
        FUNC        = #IO$_SENSEMODE,-      ; - Sensemode
        CHAN        = CHANNEL,-             ; - Channel
        IOSB        = IO_STATUS,-           ; - IOSB
        P1          = BUFFER,-              ; - User buffer supplied
        P2          = #12                   ; - Buffer length = 12
        .
        .
        .
(Check for errors)
        .
        .
        .
(Set desired characteristics bits)
        .
        .
        .
$QIOW_S -                                   ; Set new characteristics.
        FUNC        = #IO$_SETMODE,-        ; - Set Mode
        CHAN        = CHANNEL,-             ; - Channel
        IOSB        = IO_STATUS,-           ; - IOSB
        P1          = BUFFER,-              ; - User buffer address
        P2          = #12                   ; - Buffer length = 12
        .
        .
        .
(Check for errors)
        .
        .
        .
```

6.6.3 Set Mode and Sense Mode Program Example

Example 6–3 shows ways of specifying sense mode and set mode, both with and without a user buffer specified, and with user buffers of different lengths.

Example 6–3 Set Mode and Sense Mode Program Example

```
        .PSECT          IMPURE, NOEXE, NOSHR
        $IODEF

DEVICE_NAME:                                    ; Name of device
        .ASCID          /MUA0/                  ;
CHANNEL:                                        ; VMS channel to device
        .WORD           0                       ;
BUFFER: .BLKL           3                       ; Set/Sense characteristics
                                                ;  buffer
IO_STATUS:                                      ; Final I/O status
        .QUAD           0                       ;

        .PSECT          CODE, RD, NOWRT, EXE
        .ENTRY          MAIN,^M<>
        $ASSIGN_S -                             ; Assign a channel to device
                DEVNAM      = DEVICE_NAME,-     ;
                CHAN        = CHANNEL           ;
        BSBW    ERR_CHECK2                      ; Check for errors
        $QIOW_S -                               ; Get current characteristics
                FUNC        = #IO$_SENSEMODE,-; No user buffer supplied
                CHAN        = CHANNEL,-         ;
                IOSB        = IO_STATUS         ;
        BSBW    ERR_CHECK                       ; Check for errors

        $QIOW_S -                               ; Get current characteristics
                FUNC        = #IO$_SENSEMODE,-; User buffer supplied, length
                CHAN        = CHANNEL,-         ;  defaulted
                IOSB        = IO_STATUS,-       ;
                P1          = BUFFER            ;
        BSBW    ERR_CHECK                       ; Check for errors
        $QIOW_S -                               ; Get current characteristics
                FUNC        = #IO$_SENSEMODE,-; User buffer supplied, length
                CHAN        = CHANNEL,-         ;  = 8
                IOSB        = IO_STATUS,-       ;
                P1          = BUFFER,-          ;
                P2          = #8                ;
        BSBW    ERR_CHECK                       ; Check for errors
```

(continued on next page)

Example 6–3 (Cont.) Set Mode and Sense Mode Program Example

```
        $QIOW_S -                               ; Get extended characteristics
                FUNC        = #IO$_SENSEMODE,-; User buffer supplied, length
                CHAN        = CHANNEL,-       ;  = 12
                IOSB        = IO_STATUS,-     ;
                P1        = BUFFER,-          ;
                P2        = #12               ;

        BSBW    ERR_CHECK                       ; Check for errors

        $QIOW_S -                               ; Set new characteristics
                FUNC        = #IO$_SETMODE,-  ; Length defaulted
                CHAN        = CHANNEL,-       ;
                IOSB        = IO_STATUS,-     ;
                P1          = BUFFER          ;

        BSBW    ERR_CHECK                       ; Check for errors

        $QIOW_S -                               ; Set new characteristics
                FUNC        = #IO$_SETMODE,-  ; Length = 8
                CHAN        = CHANNEL,-       ;
                IOSB        = IO_STATUS,-     ;
                P1          = BUFFER,-        ;
                P2          = #8              ;

        BSBW    ERR_CHECK                       ; Check for errors

        $QIOW_S -                               ; Set extended characteristics
                FUNC        = #IO$_SETMODE,-  ; Length = 12
                CHAN        = CHANNEL,-       ;
                IOSB        = IO_STATUS,-     ;
                P1        = BUFFER,-          ;
                P2        = #12               ;

        BSBW    ERR_CHECK                       ; Check for errors

        RET

        .ENABLE LSB
ERR_CHECK:
        BLBS    IO_STATUS,ERR_CHECK2            ; Continue if good IOSB
        MOVZWL  IO_STATUS,-(SP)                 ; Otherwise, set up for stop
        BRB     10$                             ; Branch to common code

ERR_CHECK2:
        BLBS    R0,20$                          ; Continue if good status
        PUSHL   R0                              ; Otherwise, set up for stop
10$:    CALLS       #1,G^LIB$STOP               ; Stop execution

20$:    RSB

        .DISABLE LSB

        .END        MAIN
```

7 Mailbox Driver

The VMS operating system 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 much like other I/O devices, they are not hardware devices. Rather, mailboxes are a software-implemented way to perform read and write operations.

Multiport memory mailboxes function in the same way as regular mailboxes. They can also be used by processes on different processors connected to an MA780 multiport memory option.

The *Guide to VMS Programming Resources* and the *VMS System Services Reference Manual* contain additional information on the use of mailboxes.

7.1 Mailbox Operations

Table 7–1 lists the different operations that software mailboxes perform.

Table 7–1 Mailbox Read and Write Operations

Operation	Description
Receive mail	A process initiates a read request to a mailbox to obtain data sent by another process. The process reads the data if a message was previously transmitted to the mailbox.
Receive notification of mail	A process specifies that it be notified through an AST when a message is sent to the mailbox.
Send mail (without notification of receipt)	A process initiates a write request to another mailbox to transmit data to second 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)	A process initiates a write request to another mailbox to transmit data to second process. The sending process waits until the receiving process reads the data before completing the I/O operation.
Reject mail	The receiving process reads messages from the mailbox, sorts out unwanted messages, and responds only to useful messages.

7.1.1 Creating Mailboxes

To create a mailbox and assign a channel and logical name to it, a process uses the Create Mailbox and Assign Channel ($CREMBX) system service. The system enters the logical name in the job 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. A mailbox is created as either a temporary mailbox or a permanent mailbox; both types of mailboxes require privilege to create. Applications and restrictions on use of temporary and permanent mailboxes are described in the sections that follow. (See the *VMS System Services Reference Manual* for additional information on creating mailboxes.)

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.

If sufficient dynamic memory for the mailbox data structure is not available when a mailbox is created, a resource wait occurs if resource wait mode is enabled.

When a mailbox is created, a certain amount of space is specified for buffering messages that have been written to the mailbox, but they have not yet been read. The **bufquo** argument to the $CREMBX system service specifies this amount or quota. If that argument is omitted, its value defaults to the system generation parameter DEFMBXBUFQUO.

A message written to a mailbox, in the absence of an outstanding read request, is queued to the mailbox, and the size of the message (the QIO P2 argument) is subtracted from the available buffering space. After the message is read, it is added back to the available buffering space.

If a process attempts to write to a mailbox that is full or has insufficient buffering space, and if the process has resource wait enabled (which is the default case), the process is placed in miscellaneous resource wait mode until sufficient space is available in the mailbox. If resource wait is not enabled, the I/O completes with the status return SS$_MBFULL in the I/O status block (IOSB).

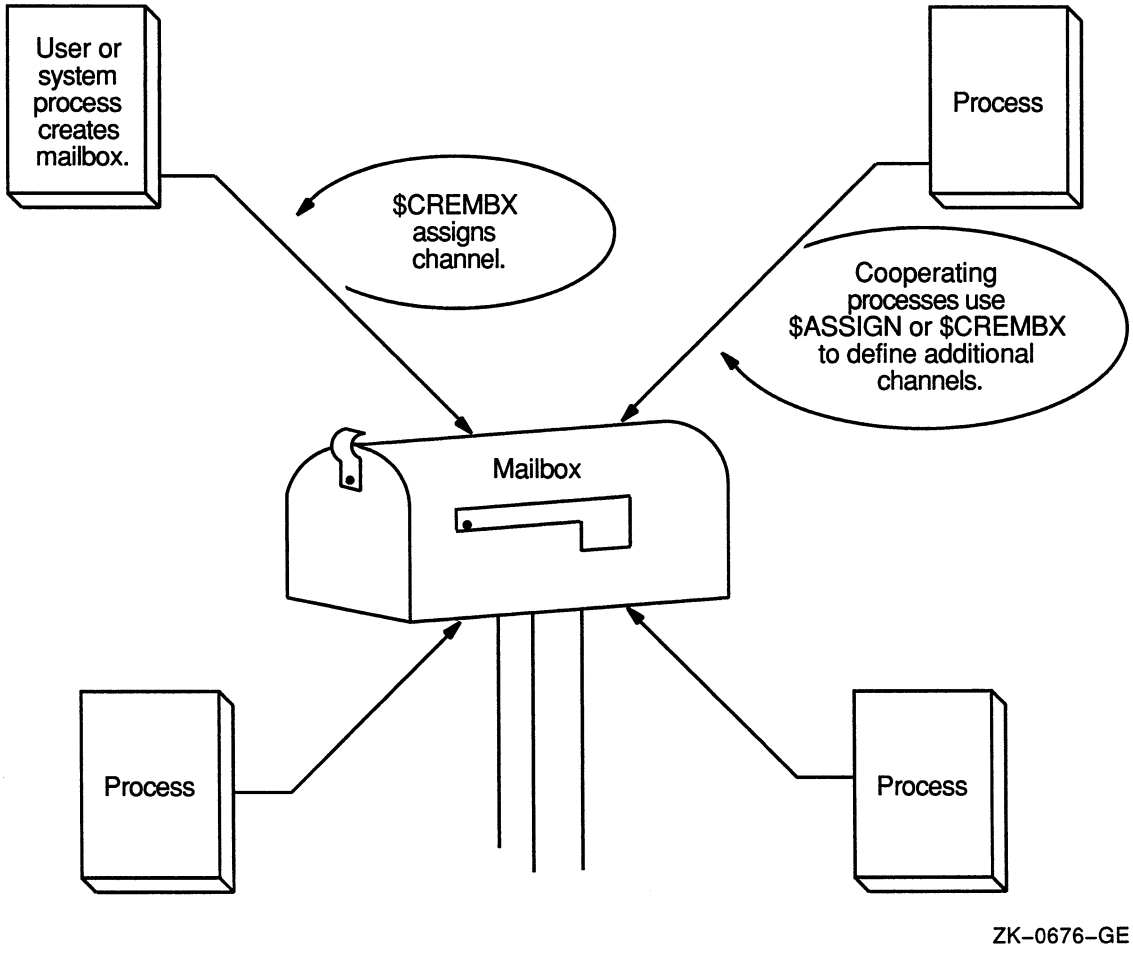
The programming example at the end of this chapter (Example 7–1) illustrates mailbox creation and interprocess communication.

7.1.2 Deleting Mailboxes

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 1. The system maintains a count of all channels and automatically deletes the mailbox when no more channels are assigned to it (that is, when the channel count reaches 0).

If a mailbox channel is deassigned, all messages sent through that channel are deleted unless the IO$M_NOW function modifier was specified with the write request.

Figure 7–1 Multiple Mailbox Channels



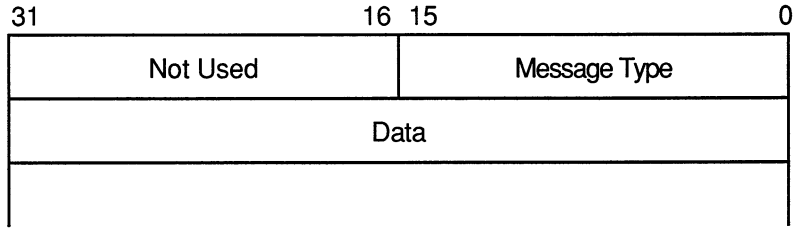
Permanent mailboxes must be explicitly deleted using the Delete Mailbox ($DELMBX) system service. An explicit deletion can occur at any time. However, the mailbox is actually deleted when no processes have channels assigned to it.

When a temporary mailbox is deleted, its message buffer quota is returned to the process that created it. (No quota charge is made for permanent mailboxes.)

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 8–2 in Section 8.2.4.

Figure 7–2 Typical Mailbox Message Format



ZK–0677–GE

7.1.4 Mailbox Protection

Mailboxes (both temporary and permanent) are protected by a code, or mask, that is similar to the code used in protecting volumes. As with volumes, four types of users (defined by UIC) can gain access to a mailbox: SYSTEM, OWNER, GROUP, and WORLD. However, only three types of access—logical I/O, read, and write—are meaningful to users of a mailbox. Thus, when creating a mailbox, you can specify logical I/O, read, and write access to the mailbox separately for each type of user. Logical I/O access is required for any mailbox operation. The set protection function modifier provides additional control of mailbox access (see Section 7.3.5).

For additional information on temporary mailboxes and mailbox protection, see the description of the $CREMBX system service in the *VMS System Services Reference Manual*.

7.2 Mailbox Driver Device Information

You can obtain information on mailbox characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *VMS System Services Reference Manual*.)

$GETDVI returns mailbox characteristics when you specify the item code DVI$_ DEVCHAR. Table 7–2 lists these characteristics, which are defined by the $DEVDEF macro.

Table 7–2 Mailbox Characteristics

Characteristic[1]	Meaning
Dynamic Bits (Conditionally Set)	
DEV$M_SHR	Device is shareable.
DEV$M_AVL	Device is available.
Static Bits (Always Set)	
DEV$M_REC	Device is record-oriented.
DEV$M_IDV	Device is capable of input.
DEV$M_ODV	Device is capable of output.
DEV$M_MBX	Device is a mailbox.

[1]Defined by the $DEVDEF macro.

DVI$_DEVCLASS and DVI$_DEVTYPE return the device class and device type names, which are defined by the $DCDEF macro. The device class for mailboxes is DC$_MAILBOX. The device type is DT$_MBX (or DT$_SHRMBX if the mailbox is a shared memory mailbox). DVI$_DEVBUFSIZ returns the buffer size, which is the maximum message size in bytes. DVI$_DEVDEPEND returns a longword field in which the two low-order bytes contain the number of messages in the mailbox. (The two high-order bytes are not used and should be ignored.)

DVI$_UNIT returns the mailbox unit number. Use of a mailbox to hold a termination message for a subprocess or a detached process requires that the parent process obtain this number to pass to the **mbxunt** argument of the $CREPRC system service.

7.3 Mailbox Function Codes

The 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 functions are used to obtain messages written by other processes. The VMS operating system provides the following mailbox function codes:

- IO$_READVBLK—Read virtual block
- IO$_READLBLK—Read logical block

- IO$_READPBLK—Read physical block

The following device- or function-dependent arguments are used with these codes:

- P1—The starting virtual address of the buffer that is to receive the message read. If P2 specifies a zero-length buffer, P1 is ignored.
- P2—The size of the buffer in bytes (limited by the maximum message size for the mailbox). A zero-length buffer may be specified. If a message longer than the buffer is read, the alternate success status SS$_BUFFEROVF is returned in the I/O status block. In such cases, the message is truncated to fit the buffer. The driver does not provide a means for recovering the deleted portion of the message.

The following function modifier can be specified with a read request:

- IO$M_NOW—Complete the I/O operation immediately with no wait for a write request from another process

Figure 7–3 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; the read request can either precede or follow the write request.

If process A issues a read request before process B issues a write request, one of two events can occur. If process A did not specify the function modifier IO$M_NOW, process A's request is queued before process B issues 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$M_NOW function modifier, the read operation is completed immediately. That is, process A's request is not queued until process B issues the write request, and no data is transferred from process B to process A. In this case, the I/O status returned to process A is SS$_ENDOFFILE.

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 message in the mailbox. The data is transferred and the I/O operation is completed immediately.

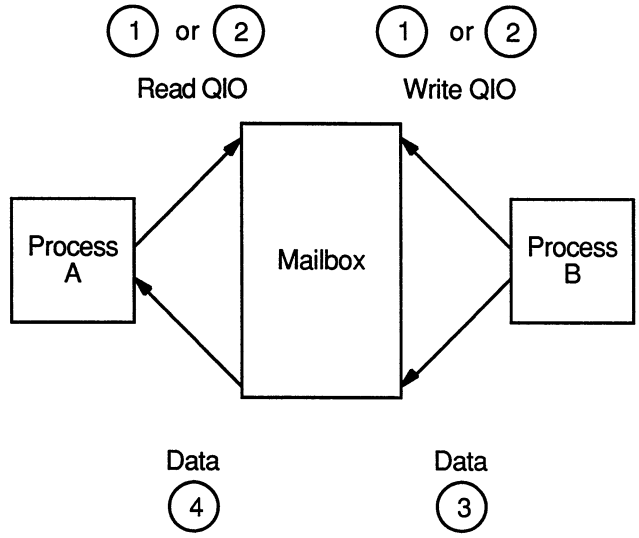
To issue the read request, process A can specify any of the read function codes; all perform the same operation.

7.3.2 Write

Write mailbox functions are used to transfer data from a process to a mailbox. The VMS operating system provides the following mailbox function codes:

- IO$_WRITEVBLK—Write virtual block

Figure 7–3 Read Mailbox



Note: Numbers indicate order of events.

ZK–0679–GE

- IO$_WRITELBLK—Write logical block
- IO$_WRITEPBLK—Write physical block

These function codes take the following device- or function-dependent arguments:

- P1—The starting virtual address of the buffer that contains the message being written. If P2 specifies a zero-length buffer, P1 is ignored.
- P2—The size of the buffer in bytes (limited by the maximum message size for the mailbox). A zero-length buffer produces a zero-length message to be read by the mailbox reader.

The following function modifiers can be specified with a write request:

- IO$M_NOW—Complete the I/O operation immediately with no wait for another process to read the mailbox message
- IO$M_NORSWAIT—If the mailbox is full, the I/O operation fails with a status return of SS$_MBFULL rather than placing the process in resource wait mode

Figure 7–4 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, a mailbox write request requires a corresponding mailbox read request (unless an error occurs), and the requests can be made in any sequence.

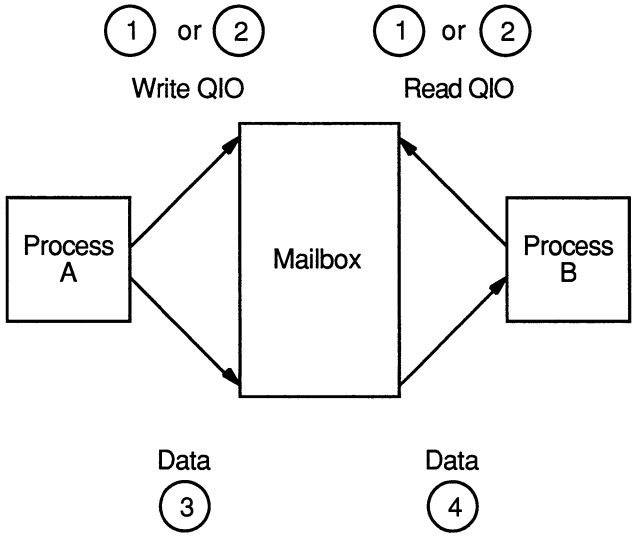
If process A issues a write request before process B issues a read request, one of two events can occur. If process A did not specify the function modifier IO$M_NOW, process A's write request is queued before process B issues a read request. When this request occurs, the data is transferred from process 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 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 function codes; all perform the same operation.

Figure 7–4 Write Mailbox



Note: Numbers indicate order of events.

ZK–0680–GE

7.3.3 Write End-of-File Message

Write end-of-file message functions 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. The VMS operating system provides the following function code:

- IO$_WRITEOF—Write end-of-file message

The following function modifier can be specified with a write end-of-file request:

- IO$M_NOW—Complete the I/O operation immediately

7.3.4 Set Attention AST

Set attention AST functions are used to specify that an AST be delivered to the requesting process when a cooperating process places an unsolicited read or write request in a designated mailbox. If a message exists in the mailbox when a request to enable a write attention AST is issued, the AST routine is activated immediately. If no message exists, the AST is delivered when a read or write message arrives. Thus the requesting process need not repeatedly check the mailbox status. You must have both logical I/O and read access to the mailbox prior to performing a set attention AST function.

The VMS operating system provides the following function codes:

- IO$_SETMODE!IO$M_READATTN—Read attention AST
- IO$_SETMODE!IO$M_WRTATTN—Write attention AST

These function codes take the following device- or 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 enabled only once; they must be explicitly reenabled after the AST has been delivered if you desire notification of the next unsolicited request. Both types of enable functions, and more than one of each type, can be set at the same time. The number of enable functions is limited only by the AST quota for the process.

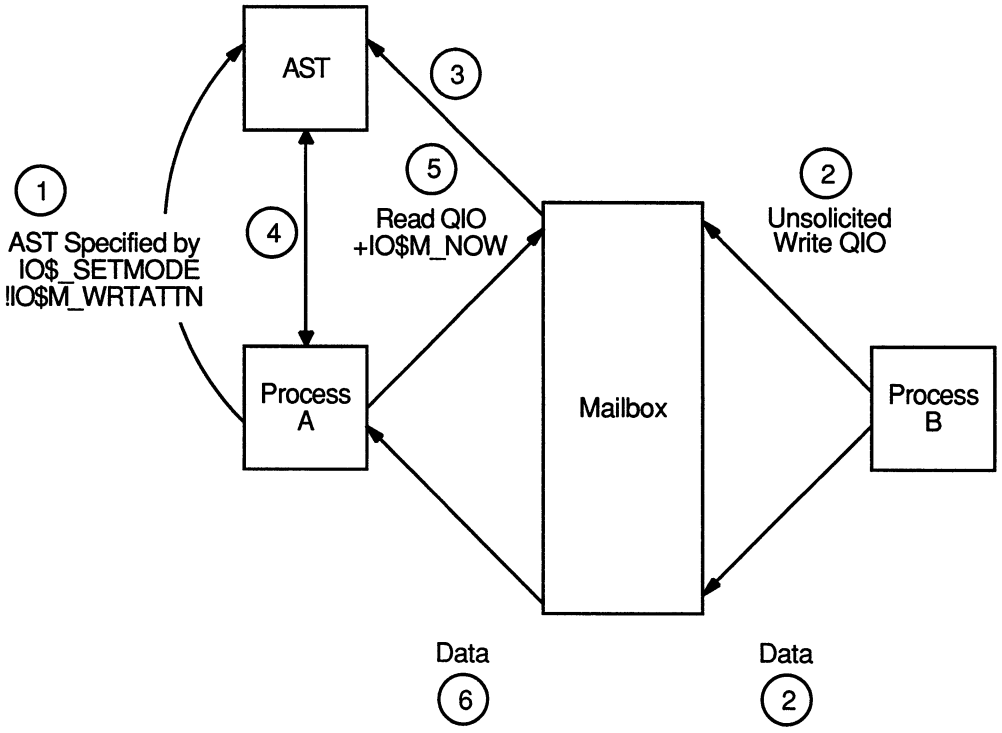
Figure 7–5 illustrates the write attention AST function. In this figure, an AST is set to notify process A when process B sends an unsolicited message.

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 might find the mailbox empty. If this request does not include the function modifier IO$M_NOW, it is queued before the next message arrives in the mailbox.

Figure 7–5 Write Attention AST (Read Unsolicited Data)



Note: Numbers indicate order of events.

ZK–0681–GE

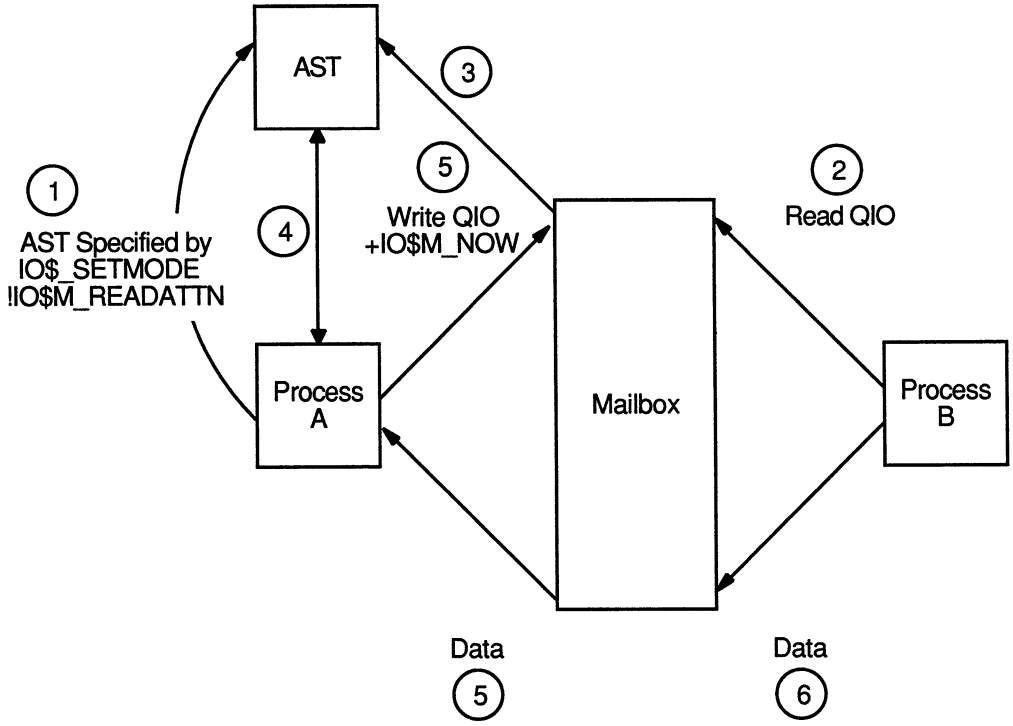
Figure 7–6 illustrates the read attention AST function. In this figure, an AST is set to notify process A when process B issues a read request for which no message is available.

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 set ASTs for read requests for 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.

Figure 7–6 Read Attention AST



Note: Numbers indicate order of events.

ZK-0682-GE

7.3.5 Set Protection

Set protection functions allow the user to set volume protection on a mailbox (see Section 7.1.4). The requester must either be the owner of the mailbox or have BYPASS privilege. The VMS operating system provides the following function code:

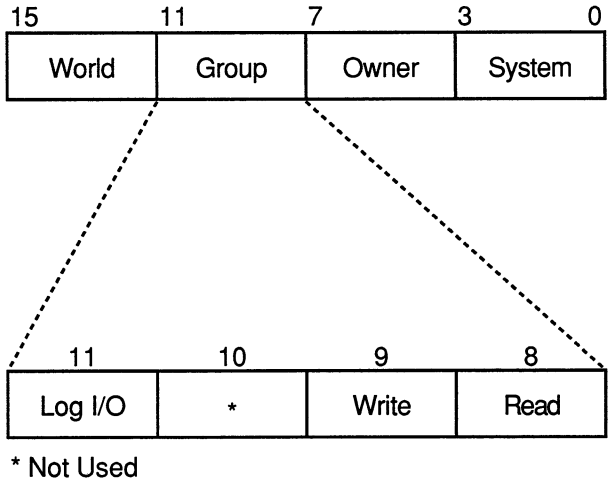
- IO$_SETMODE!IO$M_SETPROT—Set protection

This function code takes the following device- or function-dependent argument:

- P2—A volume protection mask

The protection mask specified by P2 is a 16-bit mask with four bits for each class of owner: SYSTEM, OWNER, GROUP, and WORLD, as shown in Figure Figure 7–7.

Figure 7–7 Protection Mask



Only logical I/O, read, and write functions have meaning for mailboxes. A clear (0) bit implies that access is allowed. If P2 is 0 or unspecified, the mask is set to allow all read, write, and logical operations.

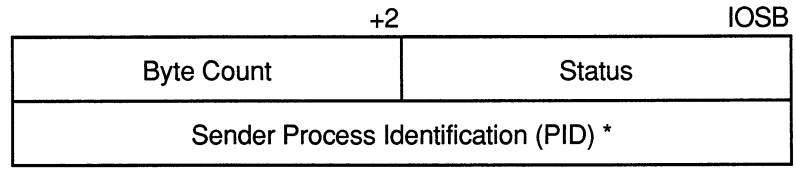
The I/O status block for the set protection function (see Figure 7–10) returns SS$_NORMAL in the first word if the request was successful. If the request was not successful, the $QIO system service returns SS$_NOPRIV and both longwords of the I/O status block are returned as zeros.

7.4 I/O Status Block

The I/O status blocks (IOSB) for mailbox read, write, and set protection QIO functions are shown in Figures 7–8, 7–9, and 7–10.

Appendix A lists the I/O status returns for these functions. In addition to these returns, the system services status returns SS$_ACCVIO, SS$_INSFMEM, SS$_MBFULL, SS$_MBTOOSML, and SS$_NOPRIV can be returned in R0. (The *VMS System Messages and Recovery Procedures Reference Manual* provides explanations and suggested user actions for both types of returns.)

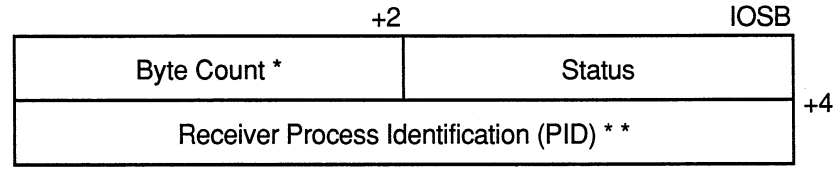
Figure 7–8 IOSB Contents - Read Function



* 0 if the sender was a system process.

ZK–0684–GE

Figure 7–9 IOSB Contents - Write Function

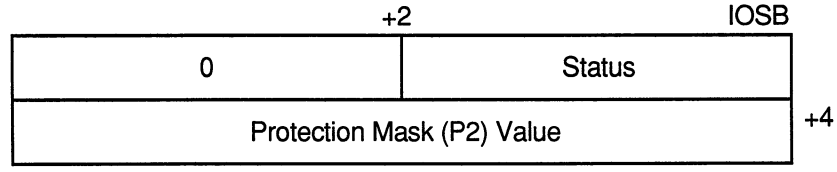


* Equals P2 buffer size if successful request.

* * 0 if IO$M_NOW was specified.

ZK–0685–GE

Figure 7–10 IOSB Contents - Set Protection Function



ZK–1201–GE

7.5 Mailbox Driver Programming Example

The following program (Example 7–1) creates a mailbox and puts mail into it; no matching read is pending on the mailbox. First, the program illustrates that if the function modifier IO$M_NOW is not used when mail is deposited, the write function waits until a read operation is performed. In this case, IO$M_NOW is specified and the program continues after the mail is left in the mailbox.

Next, the mailbox is read. If there is no mail in the mailbox, the program waits because IO$M_NOW is not specified. IO$M_NOW should be specified if there is any doubt about 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 the data that 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.

Example 7–1 Mailbox Driver Program Example

```
; ****************************************************************************
;
        .TITLE  MAILBOX DRIVER PROGRAM EXAMPLE
        .IDENT  /01/

;
; Define necessary symbols.
;
        $IODEF                          ;Define I/O function codes

;
; Allocate storage for necessary data structures.
;

;
; Allocate output device name string and descriptor.
;
DEVICE_DESCR:                           ;
        .LONG   20$-10$                 ;Length of name string
        .LONG   10$                     ;Address of name string
10$:    .ASCII  /SYS$OUTPUT/            ;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.
;
```

(continued on next page)

Example 7–1 (Cont.) Mailbox Driver Program Example

```
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

;
; 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

;
; Finally, allocate space for the I/O status quadword.
;
STATUS:                                 ;
        .QUAD   1                       ;I/O status quadword
;
; ****************************************************************
;
;                       Start Program
;
; ****************************************************************
;

;
; The program first creates a mailbox and assigns a channel to the
; process output device.  Then a message is placed in the mailbox and
; a message is received from the mailbox (the same message).  Finally,
; the program prints the contents of the mailbox on the process output
; device.
;
```

(continued on next page)

Example 7–1 (Cont.) Mailbox Driver Program Example

```
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          ;Successful mailbox creation?
        BSBW    ERROR_CHECK             ;Find out
        $ASSIGN_S -                     ;Assign channel
                DEVNAM=DEVICE_DESCR,-   ;Device descriptor
                CHAN=DEVICE_CHANNEL     ;Channel
        CMPW    #SS$_NORMAL,R0          ;Successful channel assign?
        BSBW    ERROR_CHECK             ;Find out

;
; The program now writes to the mailbox using a write request that
; includes the function modifier IO$M_NOW so that it need not wait for
; a read request to the mailbox before continuing to the next step in
; the program.
;

        $QIOW_S FUNC=#IO$_WRITEVBLK!IO$M_NOW,- ;Write message NOW
                CHAN=MAILBOX_CHANNEL,-  ;to the mailbox channel
                P1=OUT_BOX_BUFFER,-     ;Write buffer
                P2=#OUT_LENGTH          ;Buffer length
        CMPW    #SS$_NORMAL,R0          ;Successful write request?
        BSBW    ERROR_CHECK             ;Find out

;
; Read the mailbox.
;

        $QIOW_S FUNC=#IO$_READVBLK,-    ;Read the message
                CHAN=MAILBOX_CHANNEL,-  ;in the mailbox channel
                IOSB=STATUS,-           ;Define status block to
                -                       ;receive message length
                P1=IN_BOX_BUFFER,-      ;Read buffer
                P2=#IN_LENGTH           ;Buffer length
        CMPW    #SS$_NORMAL,R0          ;Successful read request?
        BSBW    ERROR_CHECK             ;Find out

;
; The program now determines how much mail is in the mailbox (this
; information is in STATUS+2) and then prints the mailbox message on
; the process output device.
;

        MOVZWL  STATUS+2,R2             ;Byte count into R2
        $QIOW_S FUNC=#IO$_WRITEVBLK,-   ;Write function to the
                CHAN=DEVICE_CHANNEL,-   ;output device channel
                P1=IN_BOX_BUFFER,-      ;Address of buffer to write
                P2=R2,-                 ;How much to write
                P4=#32                  ;Carriage control
```

(continued on next page)

Example 7–1 (Cont.) Mailbox Driver Program Example

```
;
; Finally, 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 at this point if an error was
; detected.  However, this example program simply exits.
;
ERROR_CHECK:                            ;
        BNEQ    EXIT                    ;System service failure, exit
        RSB                             ;Otherwise, return

        .END    START
```

8 Terminal Driver

This chapter describes the use of the VMS terminal driver (TTDRIVER) and the LAT port driver (LTDRIVER). The terminal driver supports the asynchronous, serial line multiplexers listed in Table 8–1. The terminal driver also supports the console terminal. The LAT port driver accommodates I/O requests from application programs, for example to make connections to remote devices, such as a printer, on a server (see Section 8.4.4).

8.1 Supported Terminal Devices

In addition to the multiplexers listed in Table 8–1, the terminal driver supports serial line interfaces that are included as part of all VAX processors. At least one such interface is always provided and is used to attach the system console terminal. This interface does not allow the setting of multiple terminal speeds, parity, or any maintenance functions, with the exception of the interface included with the VAX 8200 processor. The terminal devices supported by the VMS operating system for this interface are included in Table 8–1.

The remote command terminal, used by the DCL command SET HOST, also makes use of the features listed in Section 8.2.

Table 8–1 Supported Terminal Devices

Terminal Interface	No. of Lines	Output		Split Speed	Bus	International Modem Control
		Silo	DMA			
CXY08	8	Yes[1]	Yes	Yes	Q-bus	Full
CXA16	16	Yes[1]	Yes	Yes	Q-bus	No
CXB16	16	Yes[1]	Yes	Yes	Q-bus	No
DZQ11	4	No	No	Yes	Q-bus	No
DZQ11-CR	4	No	No	Yes	Q-bus	No
MicroVAX 2000	4	No	No	Yes	None	No
MicroVAX 3100	4	No	No	Yes	None	No
DZV11	4	No	No	No	Q-bus	No
DHQ11	8	Yes[1]	Yes	Yes	Q-bus	Full
DHU11	16	Yes	Yes	Yes	UNIBUS	Full
DHV11	8	No	Yes	Yes	Q-bus	Full
DMB32	8	No	Yes	Yes	VAXBI bus	Full

[1]Depends on whether the DHV or DHU mode is selected when the board is installed

(continued on next page)

Table 8–1 (Cont.) Supported Terminal Devices

Terminal Interface	No. of Lines	Output		Split Speed	Bus	International Modem Control
		Silo	DMA			
DHB32	16	No	Yes	Yes	VAXBI bus	Full
DSH32	8	Yes	No	Yes	MicroVAX 2000, MicroVAX 3100	No
DMF32	8	Yes	Yes[2]	Yes[2]	UNIBUS	Yes
DMZ32	24	Yes	Yes	Yes	UNIBUS	Full
DZ11	8/16	No	No	No	UNIBUS	No
DZ32	8	No	No	Limited	UNIBUS	No
LAT	[3]	No	Yes	[3]	N/A	[3]
VAX 8200 serial lines	4	No	No	No[4]	None	No
VAXstation 3100	4	No	No	Yes	None	No

[2]Lines 0 and 1.

[3]Server dependent.

[4]The VMS operating system always supports the first serial line as a console interface. The first serial line and the remaining three serial lines are also supported as user terminal interfaces at a maximum speed of 1200 baud in configurations that can include a LAT terminal interface but do not include other terminal interfaces.

8.2 Terminal Driver Features

The VMS terminal driver provides the following features:

- Input processing
  - — Command line editing and command recall
  - — Control characters and special keys
  - — Input character validation (read verify)
  - — American National Standard (ANSI) escape sequence detection
  - — Type-ahead feature
  - — Specifiable or default input terminators
  - — Special operating modes, such as NOECHO and PASTHRU
- Output processing
  - — Efficiency
  - — Limited full-duplex operation
  - — Formatted or unformatted output
- Dial-up support
  - — Modem control
  - — Hangup on logging out
  - — Preservation of process across hangups

- Miscellaneous
  - — Terminal/mailbox interaction
  - — Autobaud detection
  - — Out-of-band control character handling

8.2.1 Input Processing

The VMS terminal driver defines many terminal characteristics and read function modifiers, which provide a wide range of options to an application program. These options allow multiple levels of control over the terminal driver's input process, ranging from the default of command line editing that provides a highly flexible user interface, to the PASTHRU mode, which inhibits input process interpretation of data.

8.2.1.1 Command Line Editing and Command Recall

The terminal driver input process defines a bounded set of line editing functions. These functions are available through control keys on all keyboards, and through some special keys on certain keyboards as well. Cursor movement is provided in single-character increments (left arrow or CTRL/D, right arrow or CTRL/F), or in multicharacter increments, to beginning of the line (CTRL/H), or end of the line (CTRL/E). The terminal driver supports both insert character and overstrike character modes. The insert/overstrike mode is the terminal's default characteristic[1] at the beginning of a read operation, but it can be changed dynamically with the toggle insert/overstrike key (CTRL/A). Deletion of characters is supported in both word (CTRL/J or line feed), and to the beginning of the line (CTRL/U) increments.

When you use the terminal driver's editing functions, the following restrictions result:

- The cursor cannot be moved to a previous line after a line wrap.
- A character cannot be inserted if the insertion would force a line wrap or if a tab follows the current cursor position.
- A word cannot be deleted at the beginning of a line after a line wrap.
- The line editing function cannot be assigned to other keys.

Command recall, initiated by CTRL/B or the up arrow, returns the last line entered to the command line buffer. At this point, the line can be edited or reentered by pressing the Return key. DCL extends command recall to the last 20 commands by using the TRM$M_TM_NORECALL modifier to disable the terminal driver's recall mechanism.

Any control key that is not defined by line editing is ignored. For application programs that require control key input but do not perform QIO functions with special read modifiers, the DCL command SET TERMINAL/NOLINE_EDIT restores most of the terminal driver behavior described under VMS Versions 3.0 through 3.7.

[1] It is suggested that new users specify overstrike mode.

8.2.1.2 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, and 127 (hexadecimal 0 through 1F, and 7F); that is, all normal control characters plus DELETE. (Table B–1 lists the numeric values for all control characters.)

Some control characters are entered at the terminal by simultaneously pressing the CTRL key and a character key, such as CTRL/x. Table 8–2 lists the VMS terminal control characters. Control character echo strings (CTRL/C, CTRL/Y, CTRL/O, and CTRL/Z) can be changed on a systemwide basis (see the *VMS System Generation Utility Manual*). Special keys, such as RETURN, LINE FEED, and ESCAPE, are entered by pressing a single key.

Several of the control characters do not function as described if the SET TERMINAL/LINE_EDIT DCL command is not specified. See the *VMS DCL Dictionary* for information on line editing function keys and the SET TERMINAL command.

Table 8–2 Terminal Control Characters

Control Character	Meaning
Cancel (CTRL/C - F6[1])	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 8.4.3.2). The terminal performs a carriage-return/line-feed combination (carriage return followed by a line feed), echoes CANCEL, and performs another carriage-return/line-feed combination. If the terminal has the ReGIS characteristic or if CTRL/Y is pressed, the cancel ReGIS escape sequence is sent. Additional consequences of CTRL/C are as follows: • The type-ahead buffer is emptied. • CTRL/S and CTRL/O are reset. • All queued and in-progress write operations and all in-progress read operations are successfully completed. The status return is SS$_CONTROLC, or SS$_CONTROLY if CTRL/C is converted to CTRL/Y.
Delete character (DELETE)	Removes the last character entered from the input stream. DELETE (decimal 127 or hexadecimal 7F) is ignored if there are currently no input characters. Hardcopy 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-DELETE character is entered). Terminals that have the TT$M_SCOPE characteristic echo DELETE by removing the character.

[1] F6 on the LK201 is Interrupt/Cancel.

(continued on next page)

Table 8–2 (Cont.) Terminal Control Characters

Control Character	Meaning
Delete line (CTRL/U)	Purges current input data. When CTRL/U is entered before the end of a read operation, the current input line is deleted. (In the case of a line-wrap, CTRL/U deletes only a line at a time.) If line editing is enabled (SET TERMINAL/LINE_EDIT is specified), the data from the beginning of the line to the current cursor position is deleted.
Delete word (CTRL/J or F13) (Line feed)	Delete word before cursor. Word terminators are all control characters, space, comma, dash, period, and ! " # $ & ' ( ) + @ [\ ] ^ { | ~ / : ; <> = ? (see Appendix B).
Discard output (CTRL/O)	Discards output. Action is immediate. All output is discarded until the next read operation, the next write operation with a IO$M_CANCTRLO modifier, or the receipt of the next CTRL/O. The terminal echoes OUTPUT OFF. The current write operation (if any) and write operations performed while CTRL/O is in effect are completed with a status return of SS$_CONTROLO. A second CTRL/O, which reenables output, echoes OUTPUT ON. CTRL/C, CTRL/Y, and CTRL/T cancel CTRL/O.
End of line (CTRL/E)	Moves the cursor to the end of the line.
Exit (CTRL/Z or F10)	Echoes EXIT when CTRL/Z is entered as a read terminator. By convention, CTRL/Z constitutes end-of-file.
Interrupt (CTRL/Y)	CTRL/Y is a special interrupt or attention character that is used to invoke the command interpreter for a logged-in process. CTRL/Y can be enabled with an IO$M_CTRLYAST function modifier to an IO$_SETCHAR or IO$_SETMODE function code. The command interpreter's CTRL/Y AST handler always takes precedence over a user program's CTRL/Y AST handler. Entering CTRL/Y results in an AST to an enabled process to signify that the user entered CTRL/Y from the terminal. The terminal performs a carriage-return/line-feed combination, echoes INTERRUPT, and performs another carriage-return/line-feed combination if the AST and echo are enabled. CTRL/Y is ignored (and not echoed) if the process is not enabled for the AST. Additional consequences of CTRL/Y are as follows: • The type-ahead buffer is flushed. • CTRL/S and CTRL/O are reset. • All queued and in-progress write operations and all in-progress read operations are successfully completed with a 0 transfer count. The status return is SS$_CONTROLY. • The cancel ReGIS escape sequence is sent.
Move cursor left (CTRL/D ←)	Moves the cursor one position to the left.
Move cursor right (CTRL/F →)	Moves the cursor one position to the right.

(continued on next page)

Table 8–2 (Cont.) Terminal Control Characters

Control Character	Meaning
Move cursor to beginning of line (CTRL/H or F12) (Back space)	Moves the cursor to the beginning of the line.
Purge type ahead (CTRL/X)	Purges the type-ahead buffer and performs a CTRL/U operation. Action is immediate. If a read operation is in progress, the operation is equivalent to CTRL/U.
Recall (CTRL/B or up arrow)	Recalls last command entered. DCL extends recall to several commands.
Redisplay input (CTRL/R)	Redisplays current input. When CTRL/R is entered during a read operation, a carriage-return/line-feed combination is echoed on the terminal, and the current contents of the input buffer are 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.
Restart output (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_READSYNC, or is not currently stopped by CTRL/S.
RET (RETURN)	If used during a read (input) operation, RET echoes a carriage-return/line-feed combination. All carriage returns are filled on terminals with TT$M_CRFILL specified.
Stop output (CTRL/S)	Controls data flow; used by both terminals and the terminal driver. CTRL/S stops all data flow; the action occurs immediately with no echo. CTRL/S is also used to stop read operations. CTRL/S is meaningful only if the terminal has either the TT$M_TTSYNC characteristic or the TT$M_READSYNC characteristic.
TAB (CTRL/I)	Tabs horizontally. Advances to the next tab stop on terminals with the characteristic TT$M_MECHTAB, but the terminal driver assumes tab stops on MODULO 8 (multiples of 8) cursor positions. On terminals without this characteristic, enough spaces are output to move the cursor to the next MODULO 8 position.
Status (CTRL/T)	Displays the current time. CTRL/T also displays the current node and user name, the name of the image that is running, and information about system resources that have been used during the current terminal session.
Toggle insert/overstrike (CTRL/A or F14)	Changes current edit mode from insert to overstrike, or from overstrike to insert. The default mode (as set with SET TERMINAL/LINE_EDIT) is reset at the beginning of each line.

8.2.1.3 Read Verify

The read verify instructions provided by the terminal driver allow validation of data as each character is entered. Invalid characters are not echoed and terminate the operation. The terminal driver does not support full function field processing. Large data entry applications should use one of the DECforms, VAX FMS, or VAX TDMS layered products, which support the entire data entry environment. Section 8.4.1.4 describes the supported primitives.

8.2.1.4 Escape and Control Sequences

Escape and control sequences provide additional terminal control not furnished by the control characters and special keys (see Section 8.2.1.2). Escape sequences are strings of two or more characters, beginning with the escape character (decimal 27 or hexadecimal 1B), which 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$M_ESCAPE (see Table 8–5) is used to specify that VMS terminal lines can generate valid escape sequences. Also, the read function modifier IO$M_ESCAPE allows any read operation to terminate on an escape sequence regardless of whether TT$M_ESCAPE is set. If either TT$M_ESCAPE or IO$M_ESCAPE 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; a read buffer can contain other characters that are not part of an escape sequence, but a complete escape sequence always terminates a read operation. 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 8.5).

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. If SS$_PARTESCAPE is returned, the application program must issue enough single-character read requests, without timeout, to read the remaining characters in the escape sequence, while parsing the syntax of the rest of the escape sequence. Use of the TRM$_ESCTRMOVR item code prevents SS$_PARTESCAPE errors. No syntax integrity is guaranteed across read operations. Escape sequences are never echoed. Valid escape sequences take any of the following forms (hexadecimal notation):

```
ESC <int>...<int> <fin>      (7-bit environment)
CSI <int>...<int> <fin>      (8-bit environment)
```

The keywords in the escape sequences indicate the following:

ESC	The ESC key, a byte (character) of 1B. This character introduces the escape sequence in a 7-bit environment.
CSI	The Control Sequence Introducer, a byte (character) of 9B. This character introduces the escape sequence in a 8-bit environment.
<int>	An "intermediate character" in the range of 20 to 2F. This range includes the space character and 15 punctuation marks. An escape sequence can contain any number of intermediate characters, or none.
<fin>	A "final character" in the range of 30 to 7E. This range includes uppercase and lowercase letters, numbers, and 13 punctuation marks.

Three additional escape sequence forms are as follows:

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

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

```
ESC [ <par>...<par> <int>...<int> <fin>   (7-bit environment)
CSI   <par>...<par> <int>...<int> <fin>   (8-bit environment)
```

The keywords in the escape sequences indicate the following:

ESC	The ESC key, a byte (character) of 1B.
[	A control sequence, a byte (character) of 5B.
CSI	The Control Sequence Introducer, a byte (character) of 9B.
<par>	A parameter specifier in the range of 30 to 3F.
<int>	An "intermediate character" in the range of 20 to 2F.
<fin>	A "final character" in the range of 40 to 7E.

For example, the position cursor control sequence is ESC [Pl ; Pc H. Pl is the desired line position and Pc is the desired column position.

The user guides for the various terminals list valid escape and control sequences. For example, the *VT100 User Guide* lists the escape and control sequences for VT100 terminals.

Section 8.2.1.2 describes control character functions during escape sequences.

Table B–2 lists the valid ANSI and DIGITAL-private escape sequences for terminals that have the TT2$M_ANSICRT, TT2$M_DECCRT, TT2$M_DECCRT2, TT2$M_AVO, TT2$M_EDIT, and TT2$M_BLOCK characteristics (see Table 8–6). Table B–2 also lists assumed and selectable ANSI modes and selectable DIGITAL-private modes. Only the names of the escape sequences and modes are listed (for more information see the specific user guide for any of the various terminals). Unless otherwise noted, the operation of escape sequences and modes is identical to the particular terminals that implement these features.

8.2.1.5 Type-Ahead Feature

Input (data received) from a VMS terminal is always independent of concurrent output (data sent) to a terminal. This feature 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 Table 8–5 and Section 8.4.3).

Data entered 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 Table 8–7).

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$M_HOSTSYNC (see Table 8–5 and Section 8.4.3). If TT$M_HOSTSYNC is not set, CTRL/G (BELL) is returned to inform you that the type-ahead buffer is full. If characters are entered when the type-ahead buffer is full, the next read operation completes with a status return of SS$_DATAOVERUN. If TT$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 begin eight characters before the type-ahead buffer fills unless the TT2$M_ALTYPEAHD characteristic is set. In that case, the system generation parameter TTY_ALTALARM is used. 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 systemwide basis through use of the system generation parameter TTY_TYPAHDSZ. Terminal lines that do a large amount of bulk input should use the characteristic TT2$M_ALTYPEAHD, which allows the use of a larger type-ahead buffer specified by the system generation parameters TTY_ALTYPAHD and TTY_ALTALARM. (TTY_ALTYPAHD specifies the total size of the alternate type-ahead buffer; TTY_ALTALARM specifies the threshold at which a CTRL/S is sent.)

Certain input-intensive applications, such as block mode input terminals, can take advantage of an optimization in the driver. If a terminal has the characteristic TT2$M_PASTHRU and the read function modifier IO$M_NOECHO is specified, data is placed directly into the read buffer and thereby eliminates the overhead for moving the data from the type-ahead buffer.

8.2.1.6 Line Terminators

A line terminator is the control sequence that you type at the terminal to indicate the end of an input line. Optionally, the application 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 (see Appendix B). If line editing is enabled, the only terminators are CR, CTRL/Z, or an escape sequence. Control keys that do not have an editing function are nonfunctioning keys. If included in the request, the argument is a user-selected group of characters (see Section 8.4.1.2).

All characters are 7-bit ASCII characters unless data is input on an 8-bit terminal (see Section 8.4.1). The characteristic TT$M_EIGHTBIT determines whether a terminal uses the 7-bit or 8-bit character set; see Table 8–5. All input characters (except some special keys; see Section 8.2.1.2) are tested against the selected terminators. The input is terminated when a match occurs or your input buffer fills.

The terminal driver notifies the job controller to initiate login when it detects a carriage return terminator on a line with no current process (provided the line is not a secure server or the type-ahead feature has not been disabled). A bell character is sent when the notification occurs. A

notification character other than the bell character may be specified by setting the system generation parameter TTY_AUTOCHAR.

8.2.1.7 Special Operating Modes

The VMS terminal driver supports many special operating modes for terminal lines. (Tables 8–5 and 8–6 in Section 8.3 list these modes.) All special modes are enabled or disabled by the set mode and set characteristics functions (see Section 8.4.3).

8.2.2 Output Processing

Output handling is designed to be very efficient in the terminal driver. For example, on multiplexers that support both silo and direct memory access (DMA) ouput, the driver considers record size to decide dynamically which mode will result in the least overhead. The block size specified by the system generation parameter TTY_DMASIZE is the minimum size block that can be used in a DMA operation.

8.2.2.1 Duplex Modes

The VMS terminal driver can execute in either half- or full-duplex mode. These modes describe the terminal driver software, specifically the ordering algorithms used to service read and write requests, not the terminal communication lines.

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

In full-duplex mode, read requests (and all other requests except write 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, such as a read request and a write request at the same time. However, the driver does not execute the read request and the write request simultaneously. When it is ready to service a request, the driver decides which request—the read request or the write request—to process next.

In the VMS terminal driver, write requests usually 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 read operation, the read request becomes active when the first input character is received and echoed. Once a read request becomes active, all write requests are queued until the read completes. However, during a read operation many write requests can be executed before the first input character is entered at the terminal. Terminal lines that have the TT$M_NOECHO characteristic, or read functions that include the IO$M_NOECHO function modifier, do not inhibit write operations in full-duplex mode.

If a write function specifies the IO$M_BREAKTHRU modifier, the write operation is not blocked, even by an active read operation. IO$M_BREAKTHRU does not change the order in which write operations are queued.

When all I/O requests are entered using the Queue I/O Request and Wait ($QIOW) system service, there can be only one current I/O request at a 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.

8.2.2.2 Formatting of Output

By default, output data is subject to formatting by the terminal driver. This formatting includes actions such as wrapping, tab expansion, uppercase, and fallback conversions. Applications that do not require formatting of data can write with the IO$M_NOFORMAT modifier and thereby reduce overhead. IO$M_NOFORMAT overrides all formatting except fallback translation. Setting the PASTHRU mode (TT2$M_PASTHRU) is equivalent to writing with the noformat modifier.

Fallback conversions occur regardless of formatting mode.

8.2.2.3 SET HOST Facility and Output Buffering

The SET HOST facility emulates the VMS terminal driver in the way it writes data to the terminal by stopping the display as soon as the abort character is entered. However, the SET HOST facility behaves differently from the VMS terminal driver in that it buffers output data from the program that is executing. Occasionally, this causes a perception problem for the user when the program is aborted with a CTRL/C, CTRL/Y, or an out-of-band abort character. The user expects the program to terminate and the display to stop immediately.

CTDRIVER and RTPAD

When used between two systems running the VMS operating system, the SET HOST facility consists of two components: RTPAD on the local VAX node and CTDRIVER on the remote VAX node. Both components buffer output data to enhance performance when using wide area networks. CTDRIVER performs the initial buffering, queues the buffers for network transfer, and returns a successful write status. The user should note that the local terminal display reflects the output of the executing program after the data has been buffered and transferred over the network—not as the output buffers are filled on the remote node.

The delay between execution of an application and the display of its output can lead to several anomalies in the effects of CTRL/C, CTRL/Y, and out-of-band abort characters.

Output Line Not in Sequence Following an Abort Character

After you enter an abort character (CTRL/C, CTRL/Y or an out-of-band abort character) that causes the input or output to be aborted, it is possible to receive an additional line of output. This occurs when the application program calls $QIO (either directly or indirectly through VMS RMS or language support routines) to output data to a buffer *at the same time* the abort character is entered.

When CTDRIVER receives the abort character (CTRL/C, CTRL/Y, or an out-of-band abort character) from the network, it flushes the current output buffers and aborts any pending read operations. However, if the application program calls $QIO with a write operation when the abort character is entered, the $QIO write data is still buffered and then displayed. The data may not be the next output in sequence from the user's point of view, since all the previous output buffers in CTDRIVER were flushed and the data in them was not displayed.

When using the VMS terminal driver, the effect of an abort character on the display screen is different. The VMS terminal driver does not buffer output from the application during program execution. If the application program has just called $QIO with a write operation when the abort character is entered, then the $QIO write data is displayed. Because all write operations are sequential and do not complete until the output is actually displayed, the additional line displayed is in sequence. There is no break in the data. Normally, the user will not notice that there is an additional line.

Extra Input Prompt Following an Abort Character

For connections between systems running the VMS operating system, the CTERM protocol allows CTDRIVER to synchronize with RTPAD before displaying any more data on the terminal.

Note: Prior to VMS version 5.2, a control character entered during program execution to abort input and output could cause the system to display more than one input prompt.

If the SET HOST facility is used between systems running VMS version 5.2 and an earlier version, the extra input prompt is still displayed.

Processing Abort Characters

The abort character AST is delivered after the message describing the aborted read operation has been received. Thus, the read status should be set very shortly after the abort character AST is delivered to the application. Note, however, these are still two asynchronous events, and the application must still synchronize with the completing read operation.

Note: Prior to VMS Version 5.2, if an application had a read operation pending and had queued a CTRL/C, CTRL/Y, and out-of-band abort character AST, it was possible to queue multiple read operations unknowingly when the read operation was aborted.

Captive Command Procedures and CTRL/Y

CTDRIVER and RTPAD emulate the VMS terminal driver in that the current read operation and all pending write operations abort when CTRL/Y is entered. However, the pending write operations also include all the buffered output that occurred and that would have been output before the CTRL/Y was entered but due to the buffering was not.

The effect of the buffering can be confusing if a CTRL/Y is entered when a captive command procedure is executing. During execution of captive command procedures, DCL has a CTRL/Y pending. When this AST is delivered, DCL only reenables it; no other action is performed. In that case, if the program being executed only performs output, it appears that the program was aborted by the CTRL/Y. Actually, the program completed execution before the CTRL/Y was entered, and the CTRL/Y merely discarded all the buffered output.

8.2.3 Dial-Up Support

The VMS operating system supports modem control (for example, Bell 103A, Bell 113, or equivalent) for all supported multiplexers 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 signal) in place of the data terminal ready (DTR) signal. Most U.S. and European modems use the data terminal ready signal, which is the signal supported by the VMS operating system.

8.2.3.1 Modem Signal Control

Dial-up lines with the characteristic TT$M_MODEM are monitored periodically to detect a change in the modem carrier signals data set ready (DSR), calling indicator (RING), or request to send (RTS). The system generation parameter TTY_SCANDELTA establishes the dial-up monitoring period for multiplexers that do not support modem signal transition interrupts (see Table 8–1).

If a line's carrier signal is lost, the driver waits two seconds for the carrier signal to return. If bit 0 of the system generation parameter TTY_DIALTYPE is set to 1, the driver does not wait. Bit 0 is 0 by default for countries with Bell System standards, but that bit should be set to 1 for countries with Comite Consultatif Internationale (CCITT) standards. If the carrier signal is not detected during this time, the line is hung up. The hangup action can signal 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 a minimum of two seconds after the hangup sequence begins. The hangup sequence is not reversible. If the line hangs up, all enabled CTRL/Y and out-of-band ASTs are delivered; the CTRL/Y AST P2 argument is overwritten with SS$_HANGUP. The I/O operation in progress is canceled, and the status value SS$_HANGUP is returned in the I/O status block. DCL is responsible for process deletion after CTRL/Y is delivered. If the process is suspended, DCL cannot run, and therefore deletion cannot occur, until the process is resumed.

Note: Some systems, such as the VAXstation 3100, provide built-in serial lines using 6-pin modular jacks. These lines do not provide the minimum required modem signals. Although, the hardware may allow a dial-out connection to be established, hangup cannot be detected and process deletion will not occur on these lines.

For terminals with the TT$M_MODEM characteristic, TT$M_REMOTE reflects the state of the carrier signal. TT$M_REMOTE is set when the carrier signal changes from off to on, and cleared when the carrier signal is lost.

A line that does not have TT$M_MODEM set does not respond to modem signals or set the DTR signal. Modem signals can be set and sensed manually through use of the IO$M_MAINT function modifier (see Section 8.4.3.3).

The VMS terminal driver default modem protocol meets the requirements of the United States and of European countries. This protocol is capable of working in automatic answer mode and can also perform manually dialed outgoing calls. The protocol supports the requirements of most known international telephone networks. Enhanced modem features are used on multiplexers that support them; processor polling is not necessary. The protocol also functions in a subset mode for the multiplexers that do not support full modem signals (see Table 8–1).

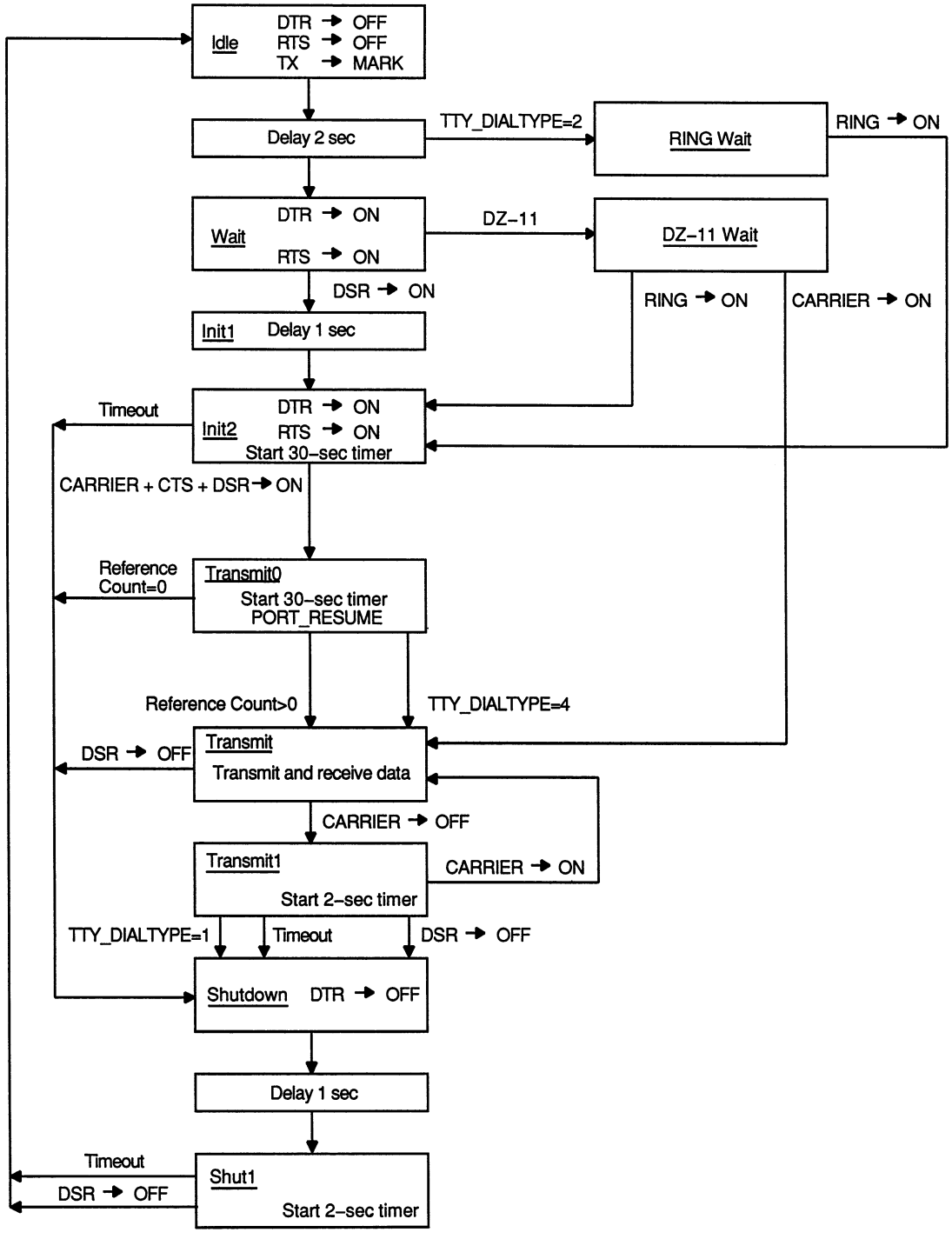
Table 8–3 lists the control and data signals used in a full modem control mode configuration (in a two-way simultaneous, symmetrical transmit mode). Figure 8–1 is a flowchart that shows a typical signal sequence for a terminal operation in this mode. The flowchart shows the states that the modem transition code goes through to detect different types of transitions in modem state. These transitions allow the driver to detect loss of lines that have been idle for several minutes. Modem states do not affect the ability of the system to transmit characters.

Set mode function modifiers are provided to allow a process to activate or deactivate modem control signals (see Section 8.4.3.3).

Bit 1 of the system generation parameter TTY_DIALTYPE enables alternate modem protocol on a systemwide basis. If bit 1 is 0 (the default), the RING signal is not used. If bit 1 is 1, the modem protocol delays setting the DTR signal until the RING signal is detected.

Remote terminal connections have a timeout feature for the security of dial-up lines. If no channel is assigned to the port within 30 seconds, or a port with an assigned channel is not allocated, the DTR signal is dropped. Such action prevents an unused terminal from tying up a line. However, there are configurations (such as a printer connected to a remote line) in which the line should not be dropped even though it is not being used interactively. To bypass the 30-second timeout, set the system generation parameter TTY_DIALTYPE to 4. (Note that if TTY_DIALTYPE is equal to 4, all dial-up lines will skip the timeout waiting for a channel to be assigned.)

Figure 8–1 Modem Control - Two-Way Simultaneous Operation



ZK–0687–GE

Table 8–3 Control and Data Signals (Full Modem Mode Configuration)

Signal	Source	MUX[1]	Meaning
Transmitted data (TxD)	Computer	All	The data originated by the computer and transmitted through the modem to one or more remote terminals.
Received data (RxD)	Modem	All	The data generated by the modem in response to telephone line signals received from a remote terminal and transferred to the computer.
Request to send (RTS)	Computer	Full	If present (ON condition), RTS directs the modem to assume the transmit mode. If not present (OFF condition), RTS directs the modem to assume the nontransmit mode after all transmit data has been transmitted.
Clear to send (CTS)	Modem	Full	Indicates whether the modem is ready (ON condition) or not ready (OFF condition) to transmit data on the telephone line.
Data set ready (DSR)	Modem	Full	If present (ON condition), DSR indicates that the modem is ready to transmit and receive; that is, the modem is connected to the line and is ready to exchange further control signals with the computer to initiate the exchange of data. If DSR is not present (OFF condition), the modem is not ready to transmit and receive. If DSR is detected, the VMS operating system initiates a 30-second timer. This ensures that the phone line will be disconnected if CARRIER is not detected.
Data channel received line signal detector (CARRIER)	Modem	All	If present (ON condition), CARRIER indicates that the received data channel line signal is within appropriate limits, as specified by the modem. If not present (OFF condition), the received signal is not within appropriate limits.
Data terminal ready (DTR)	Computer	All	If present (ON condition), DTR indicates that the computer is ready to operate, prepares the modem to connect to the telephone line, and maintains the connection after it has been made by other means. DTR can be present whenever the computer is ready to transmit or receive data. If DTR is not present (OFF condition), the modem disconnects the modem from the line.
Calling indicator (RING)	Modem	All	Indicates whether a calling signal is being received by the modem. Bit 1 of the system generation parameter TTY_DIALTYPE must be set (=1). If RING is detected, the VMS operating system initiates a 30-second timer. This ensures that the phone line will be disconnected if CARRIER is not detected.

[1]Multiplexers (All = any supported controller; Full = DZ32, DMF32, DMB32, DMZ32, DHU11, DHV11, and CXY08)

8.2.3.2 Hangup on Logging Out

By default, logging out on a line with modem signals will not break the connection. If TT2$M_HANGUP is set, modem signals are dropped when the process logs out. If TT2$M_MODHANGUP is set, no privilege is required to change the state of TT2$M_HANGUP. By setting TT2M_HANGUP, system managers can prevent nonprivileged users who are not logged in from tying up a dial-in line.

8.2.3.3 Preservation of a Process Across Hangups

Disconnectable terminals allow a connection to a physical terminal line to be broken without losing the job. The following SYSGEN command allows terminals to be disconnectable terminals:

```
SYSGEN>  CONNECT VTA0/NOADAPTER/DRIVER=TTDRIVER
```

After this command is entered, a terminal with the TT2$M_DISCONNECT characteristic logs in as VTAn:, rather than with the physical terminal name. When a terminal is set up in this manner, no input or output operations are allowed to the physical device; I/O is automatically redirected to the appropriate virtual terminal.

Following are four ways in which a terminal can become disconnected:

- Modem signals between the host and the terminal are lost.
- A user presses the BREAK key on a terminal that has the TT2$M_SECURE characteristic.
- A user issues the DCL command DISCONNECT.
- A user issues the DCL command CONNECT/CONTINUE.

After being validated as a user, you can connect to a disconnected process in either of the following ways:

- Allow the login process to make the connection.
- Issue the DCL command CONNECT.

8.2.4 Terminal/Mailbox Interaction

Mailboxes are virtual I/O devices used to communicate 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 I/O Channel ($ASSIGN) system service to associate a mailbox with one or more terminals. The terminal driver sends messages to this mailbox when terminal-related events that require the attention of the user image occur.

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 reads 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 8.4.1 and 8.4.2). (Initially, mailbox messages are enabled on all terminals. This is the default state.) 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

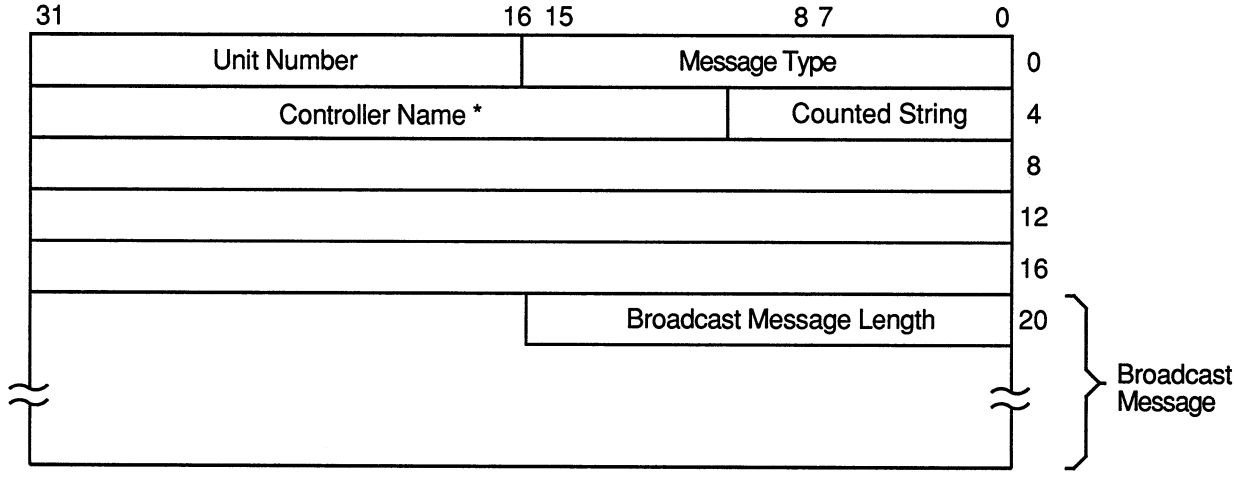
function on the last I/O exchange. Only one message is sent between read operations.

- Terminal hangup. When a remote line loses the carrier signal, it hangs up; a message is sent to the mailbox. When hangup occurs on lines that have the characteristic TT$M_REMOTE set, the line returns to local mode.
- Broadcast messages. If the characteristic TT2$M_BRDCSTMBX is set, broadcasts sent to a terminal are placed in the mailbox (this is independent of the state of TT$M_NOBRDCST).

Messages placed in the mailbox have the following content and format (see Figure 8–2):

- Message type. The codes MSG$_TRMUNSOLIC (unsolicited data), MSG$_TRMHANGUP (hangup), and MSG$_TRMBRDCST (terminal broadcast) 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.
- Message (for broadcasts).

Figure 8–2 Terminal Mailbox Message Format



* Does not include the colon (:) character.

ZK–0686–GE

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. Chapter 7 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.

8.2.5 Autobaud Detection

If you specify the /AUTOBAUD qualifier with the SET TERMINAL command, automatic baud rate detection is enabled, allowing the terminal baud rate to be set when you log in. The baud rate is set at login by pressing the Return key two or more times separated by an interval of at least one second. (Pressing a key other than Return might detect the wrong baud rate; if this occurs, wait for the login procedure to time out before continuing.) The supported baud rates are 110, 150, 300, 600, 1200, 1800, 2400, 3600, 4800, 9600, and 19200. Parity is allowed on these lines.

The autobaud function works with either even parity or no parity, but not with odd parity. If a line is set to even parity and has seven bits of data, the line automatically switches to no parity if a terminal not generating parity attempts to log in.

The SET TERMINAL qualifier /EIGHT_BIT specifies that the terminal uses eight-bit ASCII code. /NOEIGHT_BIT, which is the default, specifies seven-bit ASCII code. (If parity is specified, the parity bit is separate from the data bits.) The optimal settings for automatic baud rate detection on Digital terminals are /NOEIGHT_BIT/PARITY=EVEN or /EIGHT_BIT /NOPARITY, although automatic baud rate detection also works with other combinations, such as /NOEIGHT_BIT/NOPARITY.

Table 8–6 describes the terminal characteristic TT2$M_AUTOBAUD, which allows the baud rate to be set automatically at login.

Specifying the /FRAME qualifier with the SET TERMINAL command is not usually recommended. The terminal driver selects the frame size (the number of data bits that the device can transmit) based on how the /PARITY and /EIGHT_BIT qualifiers are set. It might be necessary to change these values if the terminal is not made by Digital.

8.2.6 Out-of-Band Control Character Handling

All control characters (0 through 1F hexadecimal) can be enabled as out-of-band characters. Typing one of these characters immediately delivers an AST to the requesting process. DCL uses this mechanism to sense whether CTRL/T has been entered. Out-of-band character options allow using the IO$M_INCLUDE function modifier to include the character in the data stream and the IO$M_TT_ABORT function modifier to abort the current input or output operation.

8.3 Terminal Driver Device Information

You can obtain information on terminal characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *VMS System Services Reference Manual.*) The sense mode function provides an alternative means to obtain terminal characteristics; see Section 8.4.5.

$GETDVI returns terminal characteristics when you specify the item codes DVI$_DEVCHAR, DVI$_DEVDEPEND, and DVI$_DEVDEPEND2. Tables 8–4, 8–5 and 8–6 list these characteristics. Terminal characteristics are normally set during system generation to any one of, or a combination of, the values listed in Table 8–5. DVI$_DEVDEPEND returns a longword field in which the three low-order bytes contain the device-dependent characteristics and the high-order byte contains the page length. Page length can have a value in the range of 0 through 255. The $DEVDEF macro defines the device-independent characteristics, the $TTDEF macro defines the device-dependent characteristics, and the $TT2DEF macro defines the extended device-dependent characteristics.

DVI$_DEVCLASS and DVI$_DEVTYPE return the device class and device type names, which are defined by the $DCDEF and $TTDEF macros, respectively. The device class for terminals is DC$_TERM. The terminal model determines the device type. For example, the device type for the VT240 is TT$_VT200_SERIES. DVI$_DEVBUFSIZ returns the page width, which can be a value in the range of 1 through 511. The driver does not accept a value of 0.

Table 8–4 Terminal Device-Independent Characteristics

Characteristic	Meaning
DEV$M_AVL	Terminal is on line and available.
DEV$M_CCL	Carriage control is enabled.
DEV$M_DET	Terminal is detached.
DEV$M_IDV	Terminal is capable of input.
DEV$M_ODV	Terminal is capable of output.
DEV$M_OPR	Terminal is enabled as an operator console.
DEV$M_REC	Device is record-oriented.
DEV$M_RTT	Terminal has remote terminal UCB extension.
DEV$M_SPL	Device is spooled.
DEV$M_TRM	Device is a terminal.
DEV$M_NET	Terminal line is allocated for VAX-DECnet use.

Table 8–5 Terminal Characteristics

Value[1]	Meaning
TT$M_CRFILL	Terminal requires fill after the Return key is pressed (the fill type can be specified by the set mode function P4 argument).
TT$M_EIGHTBIT	Terminal uses the eight-bit ASCII character set (see Appendix B). Terminals without this characteristic use the seven-bit ASCII code. In this case, the eighth bit is masked out on received characters and is 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 8.2.1.4). Escape sequences are validated for syntax.
TT$M_HALFDUP	Terminal is in half-duplex mode (see Section 8.2.2.1). All read and write requests are executed sequentially.
TT$M_HOSTSYNC	The host system is synchronized to the terminal. CTRL/Q and CTRL/S are used to control data flow and thus keep the type-ahead buffer from filling. TT$M_HOSTSYNC should always be set on LAT terminals.
TT$M_LFFILL	Terminal requires fill after the line-feed character is processed. (The fill can be specified by the set mode P4 argument.)
TT$M_LOWER	Terminal has the lowercase character set. Unless the terminal is in the PASTHRU mode or IO$M_NOFORMAT is specified, all input and echoed lowercase characters (hexadecimal 61 to 7A) are converted to uppercase if TT$M_LOWER is not set. (The character ALTMODE (decimal 125 and 126, or hexadecimal 7D and 7E) converts to ESCAPE on terminals that do not have the lowercase characteristic TT$M_LOWER set.)
TT$M_MBXDSABL	Mailboxes associated with the terminal do not receive notification of unsolicited input or hangup (see Section 8.2.3). This bit can be set by the IO$M_DSABLMBX function modifier for read requests and cleared by the IO$M_ENABLMBX function modifier for write requests.
TT$M_MECHFORM	Terminal has mechanical form feed. The terminal driver passes form feeds directly to the terminal instead of expanding to line feeds.
TT$M_MECHTAB	Terminal has mechanical tabs and is capable of tab expansion. To accomplish correct line wrapping, the terminal driver assumes there are eight spaces between tab stops.
TT$M_MODEM	Terminal line is connected to a modem. If TT$M_MODEM is set, the terminal driver automatically handles modem control. If TT$M_MODEM is not set, all modem signals are ignored. If TT$M_MODEM is set and then cleared, a hangup is declared on the terminal line if that line is in the remote state (TT$M_REMOTE is set). If DTR and RTS are set with IO$_SETMODE!IO$M_SET_MODEM!IO$M_MAINT on a nonmodem port, DTR and RTS goes off and then back on when the port is set for modem. TT$M_MODEM is not supported for LAT devices.
TT$M_NOBRDCST	Terminal does not receive any broadcast messages.
TT$M_NOECHO	Input characters are not echoed on this terminal line (see Section 8.2.1.5).
TT$M_NOTYPEAHD	Data must be solicited by a read operation. Data is lost if received in the absence of an outstanding read request (if it is unsolicited data). Disables type-ahead feature (see Section 8.2.1.5). If this characteristic is set, login attempts on this line are disabled. See Section 8.2.3.1 for information on modem signal control.

[1]Defined by the $TTDEF macro. The prefix can be TT$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 8–5 (Cont.) Terminal Characteristics

Value[1]	Meaning
TT$M_READSYNC	Read synchronization is enabled. The host explicitly solicits all read operations by entering a CTRL/Q and terminates the operation by entering a CTRL/S. TT$M_READSYNC is not applicable to LAT terminals.
TT$M_REMOTE	Dial-up characteristic is enabled. The terminal returns to local mode when a hangup occurs on the terminal line (see Section 8.2.3). This characteristic cannot be changed; it is only informational.
TT$M_SCOPE	Terminal is a video screen display (CRT terminal), for example, the VT100 or VT240.
TT$M_TTSYNC	The terminal is synchronized to the host system. Output to the terminal is controlled by terminal-generated CTRL/Q and CTRL/S. TT$M_TTSYNC is not applicable to LAT terminals unless TT$M_PASTHRU is set and TT$M_TTSYNC is disabled, in which case the LAT session is placed in PASSALL mode.
TT$M_WRAP	A carriage-return/line-feed combination should be inserted if the cursor moves beyond the right margin. If TT$M_WRAP is not set, no carriage-return/line-feed combination is sent. The VMS operating system does not support hardware-provided wrapping functions.

[1]Defined by the $TTDEF macro. The prefix can be TT$M_ or TT$V_. TT$M_ is a bit mask whose bit corresponds to the specific field; TT$V_ is a bit number.

Table 8–6 Extended Terminal Characteristics

Value[1]	Meaning
TT2$M_ALTYPEAHD	Alternate type-ahead buffer size is enabled. Use the alternate type-ahead buffer size specified during system generation (see Section 8.2.1.5). If a type-ahead buffer already exists for a terminal line, there is no effect when this characteristic is set for that line. TT2$M_ALTYPEAHD should be set prior to using the terminal, such as in the startup command procedure. You can only set TT2$M_ALTYPEAHD; this characteristic cannot be cleared until the system is rebooted.
TT2$M_ANSICRT	ANSI CRT terminal is enabled. This characteristic is set by the SET TERMINAL command. TT2$M_ANSICRT is a subset of the ANSI standard with no DIGITAL-private escape sequences (see Appendix B). It is also a subset of the VT100-family terminals (because TT2$M_ANSICRT is a subset of TT2$M_DECCRT) and the VT100. Terminals with this characteristic must provide a display of at least 24 lines, each with 80 columns.
TT2$M_APP_KEYPAD	Notifies application programs of state to set the keypad to when exiting.
TT2$M_AUTOBAUD	Automatic baud rate detection is enabled. This characteristic allows the baud rate to be set automatically when you log in. (The baud rate is set when one or more carriage returns are entered during the login procedure.) Terminals are set to a permanent speed of 9600 baud. If TT2$M_AUTOBAUD is specified, the permanent speed must not be changed while this characteristic is in use on a given terminal line. See Section 8.2.5 for additional information on automatic baud rate detection.

[1]Defined by the $TT2DEF MACRO. The prefix can be TT2$M_ or TT2$V_. TT2$M_ is a bit mask in which the bit set corresponds to the specific field; TT2$V_ is a bit number.

(continued on next page)

Table 8–6 (Cont.) Extended Terminal Characteristics

Value[1]	Meaning
TT2$M_AVO	Advanced video is enabled. This characteristic provides the terminal with blink, bold, and flashing fields as well as a full screen of 132 character lines. TT2$M_AVO is set by the SET TERMINAL command. Appendix B lists the valid escape sequences for terminals with the TT2$M_AVO characteristic.
TT2$M_BLOCK	Block mode is enabled. This characteristic is set by the SET TERMINAL command. TT2$M_BLOCK defines additional ANSI-defined and DIGITAL-private escape sequences (see Appendix B). Terminals with this characteristic are capable of local editing and block mode transmission (XON/XOFF flow control must be honored), and have protected fields. If the terminal is used for large amounts of block input, TT2$M_ALTYPEAHD should also be specified.
TT2$M_BRDCSTMBX	Mailbox broadcasts messages. Broadcast messages are sent to an associated mailbox, if one exists.
TT2$M_DECCRT	DIGITAL CRT terminal. This characteristic is set by the SET TERMINAL command for all terminals that are upward-compatible with VT100-family terminals. TT2$M_DECCRT is a superset of TT2$M_ANSICRT. Additional ANSI-defined as well as most DIGITAL-private escape sequences are allowed for terminals with this characteristic (see Appendix B); maintenance modes, VT52 mode, and the use of the LED displays are not defined by TT2$M_DECCRT. Not all VT100-family terminals implement these features. The presence of the advanced video feature cannot be assumed because it is a VT100 option. This restricts the use of graphics attributes. However, the TT2$M_AVO characteristic can be used to determine whether additional graphic attributes are available.
TT2$M_DECCRT2	DIGITAL CRT terminal. This characteristic is set by the SET TERMINAL command for all terminals that are upward-compatible with VT200-family terminals. TT2$M_DECCRT2 is a superset of TT2$M_DECCRT.
TT2$M_DIALUP	Terminal is a dial-up line. Used by LOGINOUT for the disable dial-up control.
TT2$M_DISCONNECT	Allows terminal disconnect when a hangup occurs (that is, when modem signals are lost, when the DCL commands DISCONNECT, or CONNECT/CONTINUE are entered, or when the BREAK key is pressed on a terminal that has the TT2$M_SECURE characteristic). These terminals are created as VTAn:. (See the description for the DCL command CONNECT/DISCONNECT in the *VMS DCL Dictionary*.)
TT2$M_DMA	DMA mode. This characteristic enables the use of DMA mode for asynchronous DMA multiplexers. It is ignored by non-DMA controllers.
TT2$M_DRCS	Terminal supports loadable character fonts. This characteristic is set with the DCL command SET TERMINAL/SOFT_CHARACTERS.
TT2$M_EDIT	Terminal edit. This characteristic is set by the SET TERMINAL command for all terminals that support ANSI-defined advanced editing functions. These functions include the ability to insert or delete a line and the ability to insert or delete characters in an existing line. Terminals with this characteristic are a superset of TT2$M_DECCRT. Appendix B lists the valid escape sequences for terminals with the TT2$M_EDIT characteristic.
TT2$M_EDITING	Line editing is allowed.

[1]Defined by the $TT2DEF MACRO. The prefix can be TT2$M_ or TT2$V_. TT2$M_ is a bit mask in which the bit set corresponds to the specific field; TT2$V_ is a bit number.

(continued on next page)

Table 8–6 (Cont.) Extended Terminal Characteristics

Value[1]	Meaning
TT2$M_FALLBACK[2]	Output is transformed from the eight-bit multinational character set to a seven-bit ASCII character set on terminals that do not support the eight-bit character set (see Appendix B).
TT2$M_HANGUP	Terminal hangup. Terminal lines connected through modems are hung up when a process logs out or is deleted. The state of this characteristic cannot be changed unless TT2$M_MODHANGUP is enabled or the process has either LOG_IO or PHY_IO privilege.
TT2$M_INSERT	Sets default mode for insert or overstrike at the beginning of each read operation.
TT2$M_LOCALECHO	Local echo. This characteristic is used with TT$M_NOECHO. If both characteristics are set, only terminators and special control characters are echoed. Use of this mode is restricted to command line read operations. Application programs that use the IO$M_NOECHO function modifier will not necessarily work if TT2$M_LOCALECHO is set. Local echo is also not compatible with line editing (TT2$M_EDITING).
TT2$M_MODHANGUP	Modify hangup. If specified, TT2$M_HANGUP can be modified without privilege. Otherwise, logical or physical I/O privilege is required.
TT2$M_PASTHRU	Terminal is in PASTHRU mode; all input and output data is in seven- or eight-bit binary format (no data interpretation occurs). Data is terminated when the buffer is full or when the data that is read matches the specified terminator. If the characteristic TT$M_TTSYNC is set, CTRL/S and CTRL/Q interpretation does occur.
TT2$M_PRINTER	DIGITAL CRT terminal with a local printer port.
TT2$M_REGIS	ReGIS graphics. The terminal supports the ReGIS graphics instruction set.
TT2$M_SIXEL	SIXEL graphics. The terminal supports the SIXEL graphics instruction set.
TT2$M_SECURE	For use with nonmodem, nonautobaud lines. This characteristic guarantees that no process is connected to the terminal after the BREAK key is pressed. If TT2$M_SECURE is not set, BREAK is a null key.
TT2$M_SETSPEED	Set speed. If specified, either LOG_IO or PHY_IO privilege is required to change terminal speed. TT2$M_SETSPEED is not supported for LAT devices.
TT2$M_SYSPWD	System password. This characteristic specifies that the login procedure should require the system password before the user name prompt is displayed.
TT2$M_XON	XON/XOFF control. If a set mode function is performed on a terminal in the CTRL/S state, and if TT2$M_XON is set, output is resumed. Users should note that the driver will attempt to resume stopped (XOFF) output on the line. However, restarting the output may not be successful in all cases. The XON/XOFF feature does not work on all terminals, for example, the VT220.

[1]Defined by the $TT2DEF MACRO. The prefix can be TT2$M_ or TT2$V_. TT2$M_ is a bit mask in which the bit set corresponds to the specific field; TT2$V_ is a bit number.

[2]If an attempt is made to turn on TT2$V_FALLBACK for a disconnected virtual terminal (_VTAx:) or if the Terminal Fallback Facility (TFF) has not been activated, the status code SS$_BADPARAM is returned. For more information on TFF, refer to the *VMS Terminal Fallback Utility Manual.*

8.3.1 Terminal Characteristics Categories

The set mode and set characteristics functions (see Section 8.4.3) and the DCL command SET TERMINAL are used to change terminal characteristics. The *VMS DCL Dictionary* describes the SET TERMINAL command.

To customize terminal behavior and usage, the VMS operating system divides terminal characteristics into the following categories:

- Format effectors—The following characteristics allow the user to specify terminal-dependent formatting requirements:

  TT$M_CRFILL TT$M_EIGHTBIT TT$M_LFFILL
  TT$M_LOWER TT2$M_LOCALECHO TT$M_MECHFORM
  TT$M_MECHTAB TT$M_NOECHO TT$M_SCOPE
  TT$M_WRAP

- Generic terminal capabilities—The following characteristics specify generic terminal features available to applications programs:

  TT2$M_ANSICRT TT2$M_AVO TT2$M_BLOCK
  TT2$M_DECCRT TT2$M_DECCRT2 TT2$M_DRCS
  TT2$M_EDIT TT2$M_PRINTER TT2$M_REGIS
  TT2$M_SIXEL

  Their use allows execution of these programs without knowledge of the actual terminal type. For example, a program should check for TT2$M_DECCRT rather than for VT100 or VT101.

- Protocol—The following characteristics control protocols used by the terminal:

  TT$M_ESCAPE TT$M_HALFDUP TT$M_HOSTSYNC
  TT2$M_PASTHRU TT$M_TTSYNC

- System management—The following characteristics, normally set only at system startup, allow the system manager to regulate terminal usage:

  TT2$M_ALTYPEAHD TT2$M_AUTOBAUD TT2$M_DIALUP
  TT2$M_DISCONNECT TT2$M_DMA TT2$M_HANGUP
  TT$M_MODEM TT$M_NOTYPEAHD TT2$M_MODHANGUP
  TT2$M_SECURE TT2$M_SETSPEED TT2$M_SYSPWD

- User preference—The following characteristics allow you to customize the terminal operating mode:

  TT2$M_APP_KEYPAD TT2$M_FALLBACK TT2$M_EDITING
  TT2$M_INSERT TT$M_NOBRDCST

- Miscellaneous—The following characteristics provide greater program control of terminal operations:

  TT2$M_BRDCSTMBX TT$M_MBXDSABL TT2$M_XON

8.4 Terminal Function Codes

The basic terminal I/O functions are read, write, set mode, set characteristics, sense mode, and sense characteristics. All I/O functions can take function modifiers.

8.4.1 Read

When a read function code is issued, the user-specified buffer is filled with characters from the associated terminal. The VMS operating system provides the following read function codes:

- IO$_READVBLK—Read virtual block
- IO$_READLBLK—Read logical block
- IO$_READPROMPT—Read with prompt

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

- The user buffer is full.
- The received character is included in a specified terminator mask (see Section 8.4.1.2).

The following device- or function-dependent arguments are used with the read function codes. The codes can take all six arguments (P1 through P6) on QIO requests. The descriptions for these arguments differ when itemlist read operations are performed (see Section 8.4.1.3).

- P1—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 8–7, IO$M_TIMED).
- P4—The read terminator descriptor block address (see Section 8.4.1.2).
- P5—The starting virtual address of the prompt buffer that is to be written to the terminal; for read with prompt operations using the IO$_READPROMPT function code. (This argument is specified as a value, rather than an address as in the P1 argument.)
- P6—The size of the prompt buffer that is to be written to the terminal; for read with prompt operations using the IO$_READPROMPT function code.

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, both read and write operations are performed on the specified terminal. The prompt string buffer is formatted like any other write buffer. If cursor position specifiers are supplied, they are not interpreted by the driver but passed to the terminal.

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

Depending on the terminal type and your input, the prompt string can be very simple or quite complex—from single command prompts to screen fills followed by input data. Digital recommends that prompt strings contain only one leading line feed.

In PASTHRU mode, data received from the associated terminal is placed in the user buffer as binary information without interpretation. (Prompts are not refreshed after a broadcast in PASTHRU mode.)

8.4.1.1 Function Modifier Codes for Read QIO Functions

Eight function modifiers can be specified with IO$_READVBLK, IO$_READLBLK, and IO$_READPROMPT. Table 8–7 lists these function modifiers and IO$_EXTEND, which is described in Section 8.4.1.3. All read function modifiers are supported for LAT devices.

Table 8–7 Read QIO Function Modifiers for the Terminal Driver

Code	Consequence
IO$M_CVTLOW	Lowercase alphabetic characters (hexadecimal 61 to 7A) are converted to uppercase when transferred to the user buffer or echoed. This characteristic is used only for IO$_READLBLK, IO$_READVBLK, and IO$_READPROMPT.
IO$M_DSABLMBX	The mailbox is disabled for unsolicited data.
IO$M_ESCAPE	A valid ANSI escape sequence is recognized as a valid delimiter for the read operation. The TT$M_ESCAPE characteristic is overridden by this modifier for the current read operation.
IO$M_EXTEND	This characteristic provides additional functionality for read operations (see Section 8.4.1.3). Do not specify IO$M_EXTEND with other function modifiers.
IO$M_NOECHO	Characters are not echoed 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$M_NOECHO, which inhibits all read operation echoing. Setting IO$M_NOECHO also disables line editing.
IO$M_NOFILTR	The terminal does not interpret CTRL/U, CTRL/R, or DEL. They are passed to the user. IO$M_NOFILTR explicitly disables line editing.
IO$M_PURGE	The type-ahead buffer is purged before the read operation begins.
IO$M_TIMED	The P3 argument specifies the maximum time (seconds) that can elapse between characters received from the terminal (the timeout value for the operation). Because driver timing operates on a one-second timer, a two-second timeout must be specified to guarantee a one-second wait. The timer starts when the prompt echo is started. If the read time exceeds the time specified in P3, a timeout error (SS$_TIMEOUT) is returned in the read IOSB. All input characters received before the read operation timed out are returned in the user's buffer.

(continued on next page)

Table 8–7 (Cont.) Read QIO Function Modifiers for the Terminal Driver

Code	Consequence
	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 offset to terminator in the IOSB always indicates the number of characters read. Note that the timer starts when the prompt echo is started. If a read operation is interrupted by either a broadcast write or a synchronous write request, the timer operation is restarted.
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.

8.4.1.2 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 (see Appendix B). This is the VMS RMS standard terminator set. The delete character (hexidecimal 7F) and eight-bit controls in the range 128 through 159, and 255 (hexidecimal 80 through 9F, and FF) are also terminators. If line editing is enabled, only RETURN, CTRL/Z, or an escape sequence terminates a read operation.

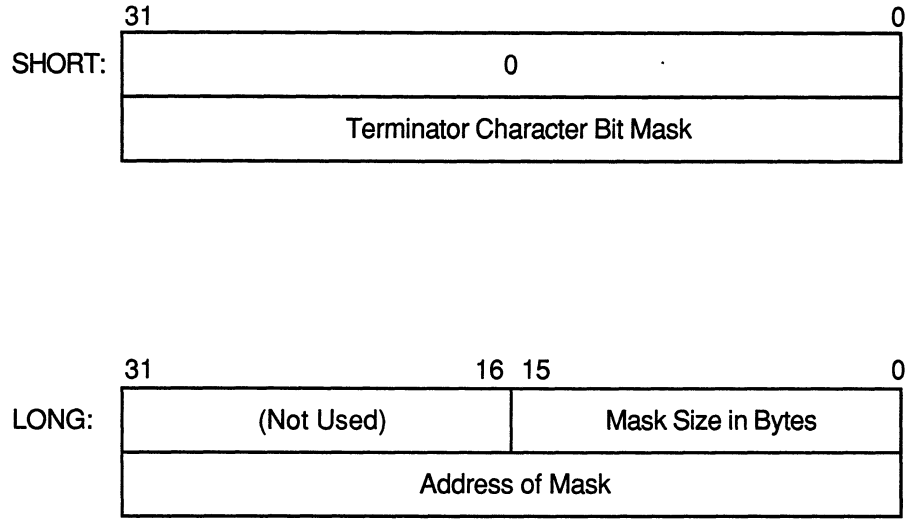
If P4 does not equal 0, it contains the address of a quadword that either specifies a terminator character bit mask or points to a location containing that mask. (Note that if P4 references an address in a MACRO program, a number sign (#) must precede the address, for example, P4=#TMASK.) The quadword has a short form and a long form, as shown in Figure 8–3. 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 one byte is acceptable. For example, a mask size of 16 bytes allows all seven-bit ASCII characters to be used as terminators; a mask size of 32 bytes allows all eight-bit characters to be used as terminators for eight-bit terminals.

If the terminator mask is all zeros, there are no specified terminators. The read operation ends when the specified number of bytes (characters) have been transferred to the input buffer.

Certain control keys will not act as terminators unless IO$M_NOFILTR is specified or the line has the TT2$M_PASTHRU characteristic (see Section 8.2.1.2.).

Figure 8–3 Short and Long Forms of Terminator Mask Quadwords



ZK–0689–GE

8.4.1.3 Itemlist Read Operations

Itemlist read operations provide expanded software features to read QIO requests. The VMS operating system provides the following combination of function code and modifier:

- IO$_READVBLK!IO$M_EXTEND—Itemlist read virtual block

No other function modifiers can be specified in an itemlist read request.

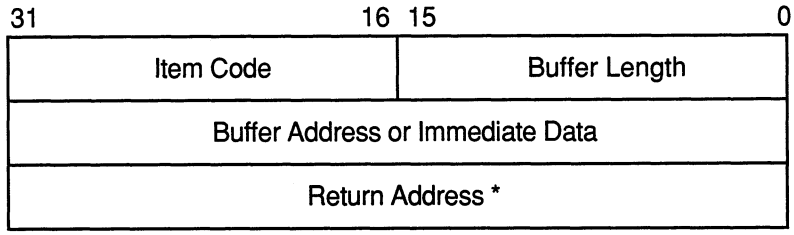
Note: Itemlist read features supported by the terminal driver are not supported by all DECNET terminal emulators.

The itemlist read function code and modifier combination takes the following device- or function-dependent arguments:

- P1—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. If required, the P2 size includes additional space for an overflow buffer to hold an escape sequence terminator (see item code TRM$_ESCTRMOVR in Table 8–8).
- P3—The access mode at which the itemlist is to be probed (optional)
- P5—The address of the itemlist buffer
- P6—The length in bytes of the itemlist buffer

P4 is not meaningful for itemlist read operations. P5 points to a series of item descriptors. Figure 8–4 shows the format for these descriptors. You cannot repeat the same item code in the same item list.

Figure 8–4 Itemlist Read Descriptor



* Must be zero.

Itemlist Read – P5 Buffer

ZK–1305–GE

Table 8–8 lists the item codes that can be specified in the first longword of the item descriptors.

Table 8–8 Item Codes for Itemlist Read Operations for the Terminal Driver

Item Code	Meaning
TRM$_ALTECHSTR	Alternate echo string. The buffer length word contains the length of the string. The data address word contains the address of the string. The alternate echo string is written to the terminal after the first character is entered. **Note: This item code for character validating read mode (TRM$K_EM_RDVERIFY) editing only.**
TRM$_EDITMODE	Extended editing modes. The immediate data longword specifies extended editing mode values. The buffer length word must be zero. The following editing modes are supported: TRM$K_EM_DEFAULT — Normal read mode. This is the default if TRM$_EDITMODE is not present in the itemlist. TRM$K_EM_RDVERIFY — Character validating read mode. See Section 8.4.1.4.
TRM$_ESCTRMOVR	Escape terminator overflow size. Specifies the number of bytes that may be used to hold an escape sequence terminator. This number should be included in P2, the buffer size argument, in addition to the space required for the data to be read. Note that this overflow area is for the terminator only; it is not available for user data. TRM$_ESCTRMOVR is useful in preventing partial escape errors, which return SS$_PARTESCAPE. This overflow buffer ensures that all the characters in an escape sequence terminator will fit in the user buffer, thus eliminating the need for additional single-character read operations.

(continued on next page)

Table 8–8 (Cont.) Item Codes for Itemlist Read Operations for the Terminal Driver

Item Code	Meaning
TRM$_FILLCHR	A two-byte value that indicates the fill and clear character for TRM$K_EM_RDVERIFY. The first byte of the immediate data longword specifies the clear character; the second byte specifies the fill character. **Note: This item code for character validating read mode (TRM$K_EM_RDVERIFY) editing only.**
TRM$_INIOFFSET	Indicates the character in the initial string where echoing starts. The immediate data longword specifies the character.
TRM$_INISTRNG	Specifies a string to preload into the read buffer (P1). The buffer length word contains the length of the string. The data longword contains the address of the string. TRM$_INISTRNG must be specified if the edit mode is TRM$K_EM_RDVERIFY, and must be the same length as specified by TRM$_PICSTRNG.
TRM$_MODIFIERS	Read modifiers. The immediate data longword contains a 32-bit value that specifies modifiers to read operations. The read operations are defined in $TRMDEF. The buffer length word must be zero. The following bits are defined: TRM$M_TM_ARROWS — The terminal interprets the left and right arrow keys (TRM$K_EM_RDVERIFY mode only). The arrow keys are not put in the buffer and do not terminate the read. TRM$_ESCTRMOVR must be greater than or equal to 5. TRM$M_TM_AUTO_TAB — This bit creates an auto-tab mode field (TRM$K_EM_RDVERIFY mode only). TRM$M_TM_CVTLOW — Lowercase alphabetic characters (hexadecimal 61 to 7A) are converted to uppercase when transferred to the user buffer or echoed. TRM$M_TM_DSABLMBX — The mailbox is disabled for unsolicited data and for receiving hangup messages. TRM$M_TM_ESCAPE — A valid ANSI escape sequence is recognized as a valid delimiter for the read operation. TRM$M_TM_NOCLEAR — Fill characters are not replaced with clear characters after a nonfill character occurs (TRM$K_EM_RDVERIFY mode only). TRM$M_TM_NOECHO — Characters are not displayed as they are entered at the keyboard. TRM$M_TM_NOEDIT — This bit inhibits advanced editing for this read operation. TRM$M_TM_NOFILTR — The terminal does not interpret DEL, CTRL/U, or CTRL/R, but passes them to you. This characteristic explicitly disables line editing.

(continued on next page)

Table 8–8 (Cont.) Item Codes for Itemlist Read Operations for the Terminal Driver

Item Code	Meaning	
	TRM$M_TM_NORECALL	This bit inhibits command recall (CTRL/B) by the terminal driver.
	TRM$M_TM_OTHERWAY	This bit sets left-justify fields to insert mode and right-justify fields to overstrike mode (TRM$K_EM_RDVERIFY mode only). TRM$M_TM_TOGGLE must equal 1.
	TRM$M_TM_PURGE	The type-ahead buffer is purged before the read operation begins.
	TRM$M_TM_R_JUST	This bit creates a right-justified field (TRM$K_EM_ RDVERIFY mode only).
	TRM$M_TM_TERM_ARROW	The read operation is terminated when the left arrow key is pressed at the left margin or when the right arrow key is pressed at the right margin (TRM$K_EM_RDVERIFY mode only). TRM$M_TM_ARROWS must be enabled.
	TRM$M_TM_TERM_DEL	The read operation is terminated when the DELETE key is pressed at the left margin (TRM$K_EM_RDVERIFY mode only).
	TRM$M_TM_TOGGLE	Enables CTRL/A to function as a toggle key between insert mode and overstrike mode (TRM$K_EM_ RDVERIFY mode only). Left-justify insert mode shifts characters to the right; right-justify insert mode shifts characters to the left. Shifted characters are not checked for validity in their new positions.
	TRM$M_TM_TIMED	TRM$_TIMEOUT specifies the maximum time (seconds) that can elapse between characters received from the terminal; that is, the timeout value for the operation. TRM$M_TM_TIMED is assumed set if TRM$_TIMEOUT is included in the itemlist.
	TRM$M_TM_TRMNOECHO	The termination character (if any) is not displayed. There is no formal terminator if the buffer is filled before the terminator is typed.

Note: All other bits must be zero.

(continued on next page)

Table 8–8 (Cont.) Item Codes for Itemlist Read Operations for the Terminal Driver

Item Code	Meaning
TRM$_PICSTRNG	Character validation string. The buffer length word contains the length of the string, which must be the same as the length specified by TRM$_INISTRNG. The data address word contains the address of the string. TRM$_PICSTRNG must be specified if the edit mode is TRM$K_EM_RDVERIFY. **Note: This item code for character validating read mode (TRM$K_EM_RDVERIFY) editing only.** The format of the character validation string is one byte per input character. Each byte is a bit mask. The following values are provided: **Value** / **Meaning** TRM$M_CV_UPPER — Uppercase alphabetic TRM$M_CV_LOWER — Lowercase alphabetic TRM$M_CV_NUMERIC — Numeric (0 - 9) TRM$M_CV_NUMPUNC — Numeric punctuation (+ - .) TRM$M_CV_PRINTABLE — Printable ASCII character TRM$M_CV_ANY — Any character If no values are set, the corresponding character specified by TRM$_INISTRNG is used. Appendix B lists the multinational character set.
TRM$_PROMPT	Specifies a prompt string. The buffer length word contains the length of the prompt. The data address word contains the address of the prompt string. See Section 8.4.1 for information on how carriage control specifiers in a prompt string are handled.
TRM$_TERM	The buffer length word determines the format of the nondefault terminator mask. If the buffer length word is zero, then the data longword is used as a short form mask. If the buffer length word is nonzero, then a mask n bytes long is available at the specified address.
TRM$_TIMEOUT	Read timeout. See the description of IO$M_TIMED in Table 8–7.

8.4.1.4 Read Verify Function

When using the read verify function, the terminal driver performs input validation based on character attributes.[1] Validation is performed one character at a time as data is entered. Invalid characters are not echoed, and cause the read operation to complete. It is then up to the application program to handle the error appropriately.

The initial string describes the initial contents of the input field. This string may consist of data and marker characters. The clear character is displayed on the screen for each occurrence of the fill character in the initial string buffer.

The picture string is a string of bytes where each byte corresponds to one character of the field being entered. Each byte specifies a mask of legal character types for that character position. If the byte is left as zero, then that position is a marker character, and the character from the initial string is echoed for that position.

[1] Read verification bypasses the optionally specified termination mask (TRM$_TERM).

For left-justified fields, the prompt data is output to the terminal, followed by an optional number (TRM$_INIOFFSET) of initial string characters. Leading marker characters are always output following the prompt, leaving the cursor at the leftmost data position. As each character is entered, it is validated and then echoed, advancing the cursor position. Additional marker characters are skipped as they are encountered. If an input character fails the validation, the read operation is completed with the invalid character as the terminator.

For right-justified fields, the prompt is output and is followed by the initial string. (In general, TRM$_INIOFFSET is set to the length of TRM$_INISTRNG for right-justified fields.) The cursor position remains one position to the right of the initial string. For proper operation, right-justified fields cannot have mixed picture definitions. After each character is input, the entire prompt and input fields are output. Therefore, the prompt should include a cursor positioning escape sequence.

The definition of full field is different for left- and right-justified read operations. For left-justified fields, full field is detected when the character corresponding to the last nonmarker position in the picture string has been entered. For right-justified fields, full field is detected when a character other than the fill character is shifted into the leftmost, nonmarker position in the field.

If the modifier TRM$M_TM_AUTO_TAB is set in TRM$_MODIFIERS, then detection of a full field terminates the read operation. In the event of autotab termination, the terminator character in the IOSB is null. If the autotab option is not selected, then termination occurs when one more character is typed to a full field. Applications can detect this condition when the terminating character index is one character beyond the end of the field. The extra character is reported as the terminator. In a left-justified field the IOSB index to the terminator is zero-based; in a right-justified field this index is one-based.

If a read verify function is interrupted by an asynchronous write operation, the read verify is completed with status SS$_OPINCOMPL.

No line editing functions other than the delete character function are supported for read verify.

8.4.2 Write

Write operations display the contents of a user-specified buffer on the associated terminal. The VMS operating system provides the following write I/O functions, which are listed with their function codes:

- IO$_WRITEVBLK—Write virtual block
- IO$_WRITELBLK—Write logical block
- IO$_WRITEPBLK—Write physical block

The write function codes can take the following device- or 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.)
- P4—carriage control specifier except for write physical block operations (Write function carriage control is described in Section 8.4.2.2.)

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.

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 occur every eight characters or positions.

CTDRIVER and Buffered Output

CTDRIVER, a component of the SET HOST facility, buffers output from remote terminals in order to package multiple output requests into a single network transfer. As a result, control is returned early to the user with a status of SS$_NORMAL when the output buffer has been filled and successfully queued.

Note that this output might not be displayed if the user enters an abort character or a CTRL/O.

8.4.2.1 Function Modifier Codes for Write QIO Functions

Five function modifiers can be specified with IO$_WRITEVBLK, IO$_WRITELBLK, and IO$_WRITEPBLK. Table 8–9 lists these function modifiers. All write function modifiers are supported for LAT devices.

Table 8–9 Write QIO Function Modifiers for the Terminal Driver

Code	Consequence
IO$M_BREAKTHRU	Allows breakthrough read regardless of the current active state.
IO$M_CANCTRLO	Turns off CTRL/O (if it is in effect) before the write operation. Otherwise, the data cannot 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 you to specify write functions without interpretation or format; in effect, the terminal line is in a temporary PASTHRU mode.

(continued on next page)

Table 8–9 (Cont.) Write QIO Function Modifiers for the Terminal Driver

Code	Consequence
IO$M_REFRESH	If a read operation is interrupted by a write operation (by either a write breakthrough[1] or any other type of write), the terminal displays the current read data when the read function is restarted.

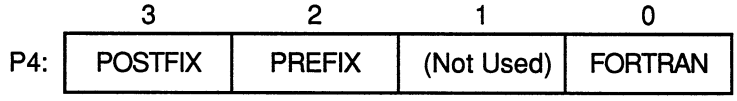
[1]Any interruption caused by the execution of the $BRDCST or the $BRKTHRU system service broadcasting messages to terminals is referred to as a "write breakthrough."

8.4.2.2 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 8–5 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 8–10 lists the possible byte 0 values (in hexadecimal) and their meanings.

Figure 8–5 P4 Carriage Control Specifier



ZK–0690–GE

Table 8–10 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: carriage-return/line-feed combination, print buffer contents, return[1])
30	0	Double-space carriage control. (Sequence: carriage-return/line-feed combination, carriage-return/line-feed combination, print buffer contents, return[1])
31	1	Page eject carriage control. (Sequence: form feed, print buffer contents, return)
2B	+	Overprint carriage control; allows double printing for emphasis or special effects. (Sequence: print buffer contents, return)
24	$	Prompt carriage control. (Sequence: carriage-return/line-feed combination, print buffer contents)
All other values		Same as ASCII space character: single-space carriage control

[1]A carriage-return/line-feed combination is a carriage return followed by a line feed.

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 as follows:

1 Prefix carriage control

2 Print

3 Postfix carriage control

The prefix and postfix bytes, although interpreted separately, use the same encoding scheme. Table 8–11 shows this encoding scheme in hexadecimal.

With several exceptions, Figure 8–6 shows the prefix and postfix hexadecimal coding that produces the carriage control functions listed in Table 8–10. Prefix and postfix coding provides an alternative way to achieve these controls.

In the first example in Figure 8–6, the prefix/postfix hexadecimal coding for a single-space carriage control (carriage-return/line-feed combination, 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)
```

Table 8–11 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 (NULL).
0		1–7F		Bits 0 through 6 are a count of carriage-return/line-feed combinations.
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 (seven-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 (eight-bit character set; see Appendix B).

8.4.3 Set Mode

Set mode operations affect the operation and characteristics of the associated terminal line. The VMS operating system provides two types of set mode functions: set mode and 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.[1] The following function code is provided:

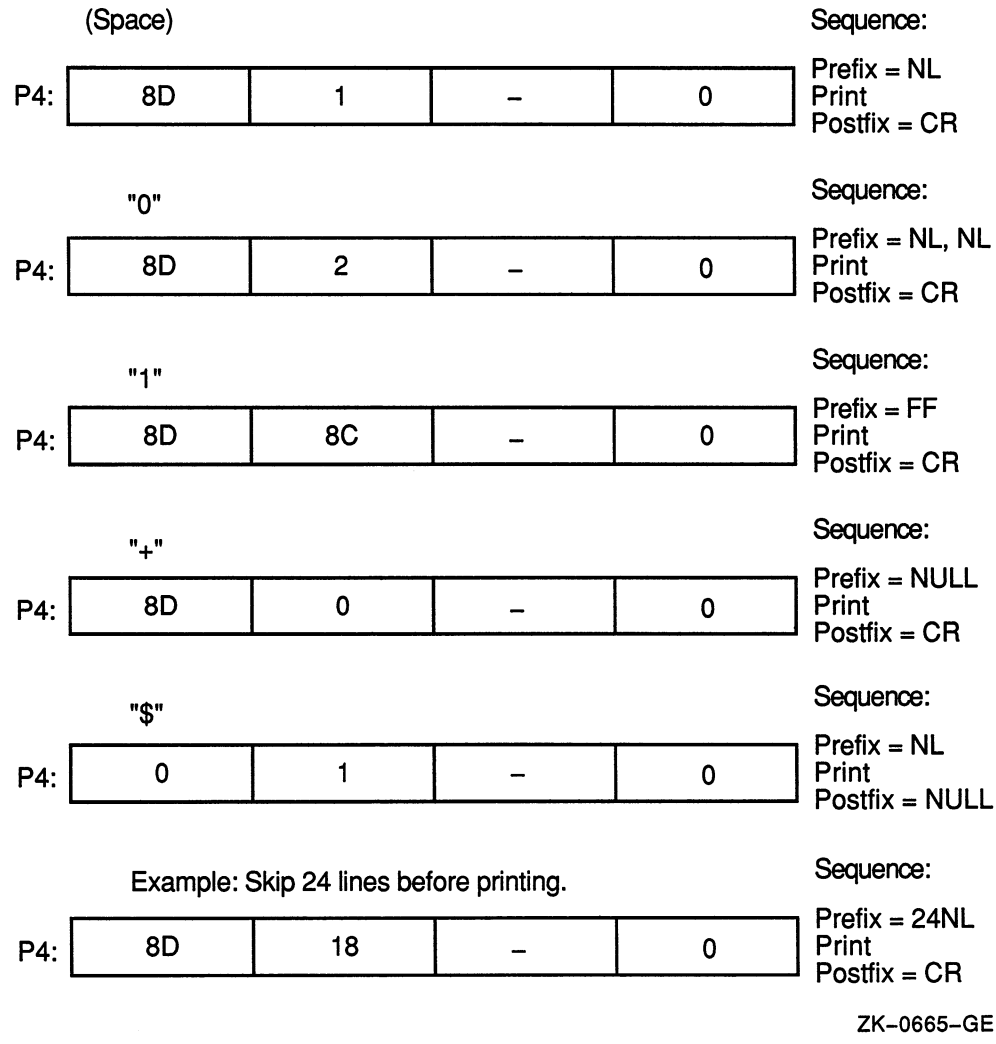
- IO$_SETMODE

The set characteristics function 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. The following function code is provided:

- IO$_SETCHAR

[1] If you do not have LOG_IO or PHY_IO privilege, the terminal driver does not accept a set mode request to a terminal that does not have the extended terminal characteristic TT2$M_SETSPEED—even if no request for a change of speed is made. Privilege is not required if TT2$M_SETSPEED is set but no attempt to change the speed is made.

Figure 8–6 Write Function Carriage Control (Prefix and Postfix Coding)

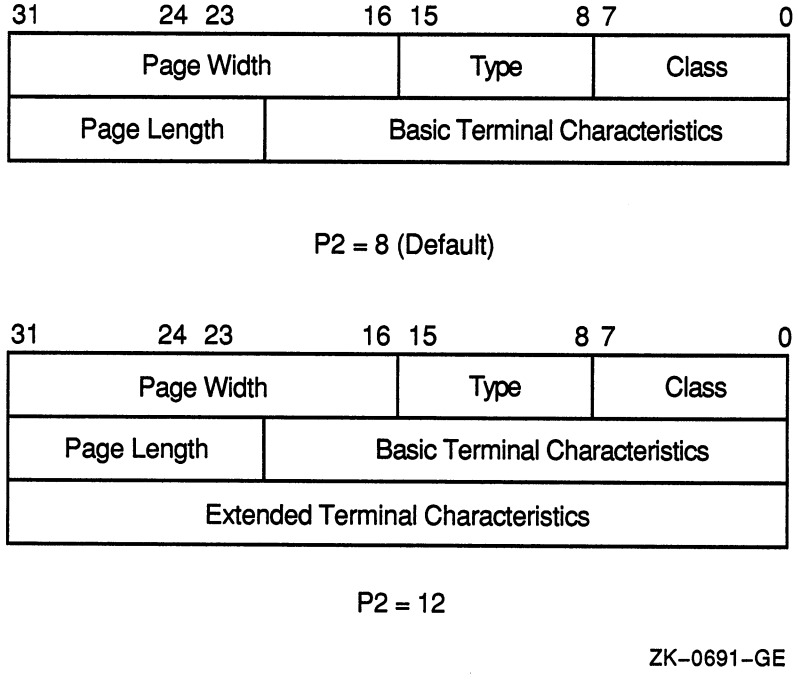


The set mode and set characteristics functions take the following device- or function-dependent arguments if no function modifiers are specified:

- P1—Address of characteristics buffer
- P2—Length of characteristics buffer (default length is 8 bytes)
- P3—Speed specifier (bits 0 through 7 = transmit; 8 through 15 = receive)
- 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 variable length block, as shown in Figure 8–7. With the exception of terminal characteristics, the contents of the block are the same for both the set mode and set characteristics functions.

Figure 8–7 Set Mode and Set Characteristics Buffers



In the buffer, the device class is DC$_TERM, which is defined by the $DCDEF macro. The terminal type is defined by the $TTDEF macro, for example, TT$_LA36. The page width is a value in the range of 1 through 511. The page length is a value in the range of 0 through 255. Table 8–5 lists the values for terminal characteristics. Table 8–6 lists the extended terminal characteristics. Characteristics values are defined by the $TTDEF and $TT2DEF macros.

Note: Make sure that the selected device is a terminal before performing any set mode function, particularly when using SYS$INPUT or SYS$OUTPUT.

The P3 argument defines the device speed, such as TT$C_BAUD_300. The low eight bits specify the transmit speed, and the high eight bits specify the receive speed. If no receive speed is specified, the indicated transmit speed is used for both transmitting and receiving. If neither the transmit nor the receive speed is specified (P3 = 0), the baud rate is not changed. The terminal driver ignores the receive speed bits for interfaces that do not support split-speed operation. While speeds up to 19.2K baud can be specified, not all controllers support all speed combinations. Refer to

the associated hardware documentation to determine which speeds are supported by your controller.

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 carriage return. Bits 8 through 15 specify the number of fill characters used after a line feed.

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 8–5.

Several parity flags can be specified in the P5 argument:

- TT$M_ALTRPAR—Alter parity. If set, check the state of TT$M_PARITY and TT$M_ODD and, if indicated, change the parity. Otherwise, ignore these bits. TT$M_ALTRPAR is not supported for LAT devices.
- TT$M_PARITY—Enable parity on terminal line if set, disable if clear.
- TT$M_ODD—Parity is odd if set.
- TT$M_ALTDISPAR—Alter dismiss parity errors. If set, check the state of TT$M_DISPARERR.
- TT$M_DISPARERR—Dismiss parity errors. If this mode is set, input errors with a parity error flagged are discarded and no error is reported.

Note: If parity is enabled, the DZ11 generates a parity check bit to detect parity mismatch. Unless TT$M_DISPARERR is enabled, parity errors that occur during an I/O read operation are fatal to the operation. Parity errors that occur on input characters (that is, keys pressed on the keyboard) when no I/O operation is in progress might result in a character loss.

- TT$M_BREAK—Generate a break if set. The break is in effect until this bit is turned off. TT$M_BREAK is supported by the LTDRIVER for terminal servers that support the break capability, such as the DECserver 200 and DECserver 500. However, in the case of LAT terminals, the terminal server controls the duration of the break.
- TT$M_ALTFRAME—If set, the four low-order bits of P5 become the frame size. Note that the frame size is for data bits only and is exclusive of parity. TT$M_ALTFRAME is not supported for LAT devices.

To take the existing parity settings, modify them, and use them in the set mode or set characteristic function, move the byte starting at the second nibble of the buffer that is going to be used in the P5 argument. For example, the following instructions change the parity from even to odd:

```
insv    iosb+6, #4, #8, flags
bisl    #tt$m_altrpar!tt$m_odd!tt$m_parity, flags
```

The following instruction then resets the parity to its original state:

```
bicl    #tt$m_odd!tt$m_parity, flags
```

See Section 8.2.5 for information about the SET TERMINAL/FRAME command.

Application programs that change terminal characteristics should perform the following steps:

1. Use the IO$_SENSEMODE function to read the current characteristics.
2. Modify the characteristics.
3. Use the set mode function to write back the results.
4. If the characteristic is intended to be reset when the image exits, the application must perform this operation.

Failure to follow this sequence will result in clearing any previously set characteristic.

Two stop bits are used only for data rates less than or equal to 150 baud; higher data rates default to one stop bit.

The set mode and set characteristics functions can take the enable CTRL/C AST, enable CTRL/Y AST, enable out-of-band AST, hangup, set modem, broadcast, and loopback function modifiers that are described in the next several sections.

Note: If an attempt is made to turn on TT2$V_FALLBACK for a disconnected virtual terminal (_VTAx:) or if the Terminal Fallback Facility has not been activated, the status code SS$_BADPARAM will be returned. For more information on TFF, refer to the *VMS Terminal Fallback Utility Manual*.

8.4.3.1 Hangup Function Modifier

The hangup function disconnects a terminal that is on a dial-up line. (Dial-up lines are described in Section 8.2.3.) The following combinations of function code and modifier are provided:

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

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

Note: For remote terminals, the hangup function breaks the network connection to the local system ending the remote terminal session.

8.4.3.2 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 you press CTRL/C or CTRL/Y. The following 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

These function code modifier pairs take the following device- or 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 enabling is in effect, pressing CTRL/C or CTRL/Y gains the attention of the enabling process (see Table 8–2).

Enable CTRL/C and CTRL/Y AST are one-time enabling function modifiers. After the AST occurs, it must be explicitly reenabled by one of the two function code combinations before an AST can occur again. This function code is also used to disable the AST. The function is subject to AST quotas.

You can have more than one CTRL/C or CTRL/Y enabled; pressing CTRL/C, for example, results in the delivery of all CTRL/C ASTs. ASTs are queued and delivered to the user process on a first-in/first-out basis for each access mode. However, ASTs are processed in the reverse order of the CTRL/C AST or CTRL/Y AST requests that have been issued to the terminal driver (on a last-in/first-out basis).

If no enable CTRL/C AST is present, the holder of an enable CTRL/Y AST receives an AST when CTRL/C is pressed; carriage-return/line-feed combination, ^Y, and RETURN are echoed.

Figure 8–9 shows the relationship of CTRL/C and CTRL/Y with the out-of-band function. If CTRL/C or CTRL/Y is an enabled out-of-band character, any out-of-band ASTs specified for this character are delivered. If the IO$M_INCLUDE function modifier is included in the out-of-band AST request for this character, an enabled CTRL/C or CTRL/Y AST is also delivered.

Enable CTRL/C AST requests are flushed by the Cancel I/O on Channel ($CANCEL) system service. Enable CTRL/Y AST requests are flushed by the Deassign I/O Channel ($DASSGN) system service.

CTRL/Y is normally used to gain the attention of the command interpreter and to input special commands such as DEBUG, STOP, and CONTINUE. Programs that are run from a command interpreter should not enable CTRL/Y. Because ASTs are delivered on a first-in/first-out basis, the command interpreter's AST routine gets control first, and might not allow the program's AST to be delivered at all. Programs that require the use of CTRL/Y should use the LIB$DISABLE_CTRL RTL routine to disable DCL recognition of CTRL/Y.

Section 8.2.1.2 describes other effects of CTRL/C and CTRL/Y.

8.4.3.3 Set Modem Function Modifier

The set modem function modifier is used in maintenance operations to allow a process to activate and deactivate modem control signals. Both set mode and set characteristics functions can take the set modem function modifier. The following combinations of function code and modifier are provided:

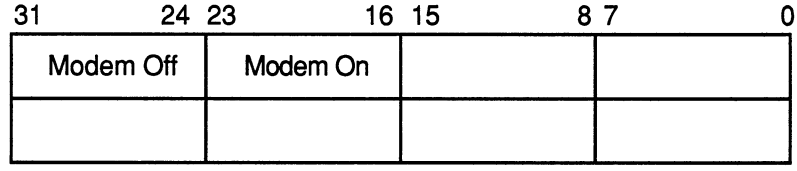
- IO$_SETMODE!IO$M_SET_MODEM!IO$M_MAINT
- IO$_SETCHAR!IO$M_SET_MODEM!IO$M_MAINT

These function code modifier pairs take the following device- or function-dependent argument:

- P1—The address of a quadword block that specifies which modem control signals to activate or deactivate

Figure 8–8 shows the format of this block.

Figure 8–8 Set Mode P1 Block



ZK-0692-GE

The modem on and modem off fields, in combination or separately, can specify one or more of the following values:

- TT$M_DS_RTS—Request to send (RTS)
- TT$M_DS_DTR—Data terminal ready (DTR)
- TT$M_DS_SECTX—Transmitted backward channel data (Sec Txd)

The $TTDEF macro defines which of these values the modem on and modem off fields specify. These values can only be specified if the terminal characteristic TT$M_MODEM is not set. Otherwise, an error (SS$_ABORT) will result.

Note 1: The set modem function is not supported for remote terminals. The status SS$_DEVREQERR is returned in the I/O status block.

Note 2: Because the DMF32 does not provide the secondary transmitted data signal (Sec Txd), the driver sets the secondary request to send the signal. Users should connect a jumper cable between pins 14 and 19 on the DMF32.

8.4.3.4 Loopback Function Modifier

The loopback function modifier is used in maintenance operations to place the terminal line in a hardware loopback mode. Data transmitted to a line in this mode is returned as receive data. If the controller does not support loopback mode or the terminal line has the TT$M_MODEM characteristic set, an error status (SS$_ABORT) is returned. Both set mode functions can take the loopback function modifier.

Note: The loopback function is not supported for remote terminals. The status SS$_DEVREQERR is returned in the I/O status block.

The following combinations of function code and modifier are provided:

- IO$_SETMODE!IO$M_LOOP!IO$M_MAINT
- IO$_SETCHAR!IO$M_LOOP!IO$M_MAINT

Data transmitted in the loopback mode should only be written in records less than or equal to the size of the type-ahead buffer (see Section 8.2.1.5). Programs that use the loopback function modifier should incorporate a one-second delay to allow the controller to enable the loopback mode after the request is posted. Write requests should also include the IO$M_NOFORMAT function modifier to prevent the terminal driver from formatting input or output data.

Note: The serial line interfaces for the VAX 8200 processor implement an internal loopback bus that is common to all four serial lines. The hardware allows all serial lines operating in loopback mode to transmit data to the bus at the same time. If more than one serial line writes data to the bus, all of the transmitted data is combined and made available to the receiving end of those same serial lines. Thus, the received data may be different from the transmitted data if more than one serial line is operating in loopback mode at the same time. To prevent receiving such spurious data, you must not operate multiple serial lines in loopback mode.

The VMS operating system provides another function modifier to reset a terminal line previously placed in loopback mode. The following combinations of function code and modifier are provided:

- IO$_SETMODE!IO$M_UNLOOP!IO$M_MAINT
- IO$_SETCHAR!IO$M_UNLOOP!IO$M_MAINT

Programs that use the unloop function modifier should incorporate a one-second delay to allow the controller to reset the loopback mode after the request is posted.

Note: IO$M_LOOP and IO$M_UNLOOP are not supported for LAT devices.

8.4.3.5 Enable Out-of-Band AST Function Modifier

The enable out-of-band AST function modifier requests that the terminal driver queue an AST for the requesting process when you enter any one of 32 control characters. The following combinations of function code and modifier are provided:

- IO$_SETMODE!IO$M_OUTBAND—Enable out-of-band AST
- IO$_SETCHAR!IO$M_OUTBAND—Enable out-of-band AST

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

- P1—Address of the AST service or 0 if the AST entered on this channel is to be canceled. (The AST parameter will be the out-of-band character.)
- P2—Address of a character mask with the same format as the short form terminator mask (see Section 8.4.1.2).
- P3—Access mode to deliver AST (maximized with the caller's access mode).

The IO$_SETMODE!IO$M_OUTBAND function can optionally take the following function modifiers:

- IO$M_INCLUDE—Include the character typed in the data stream.
- IO$M_TT_ABORT—Allow current read and write operations to be aborted. (The IOSB for aborted operations returns the status SS$_CONTROLC.)

If an out-of-band AST is in effect, pressing any control character specified in the P2 mask gains the attention of the enabling process. Figure 8–9 shows the relationship of the out-of-band function with some of the control characters.

You can have only one out-of-band AST enabled per channel.

Out-of-band ASTs are repeating ASTs; they continue to be delivered until specifically disabled. Out-of-band AST enables are flushed by the Cancel I/O on Channel ($CANCEL) system service.

8.4.3.6 Broadcast Function Modifier

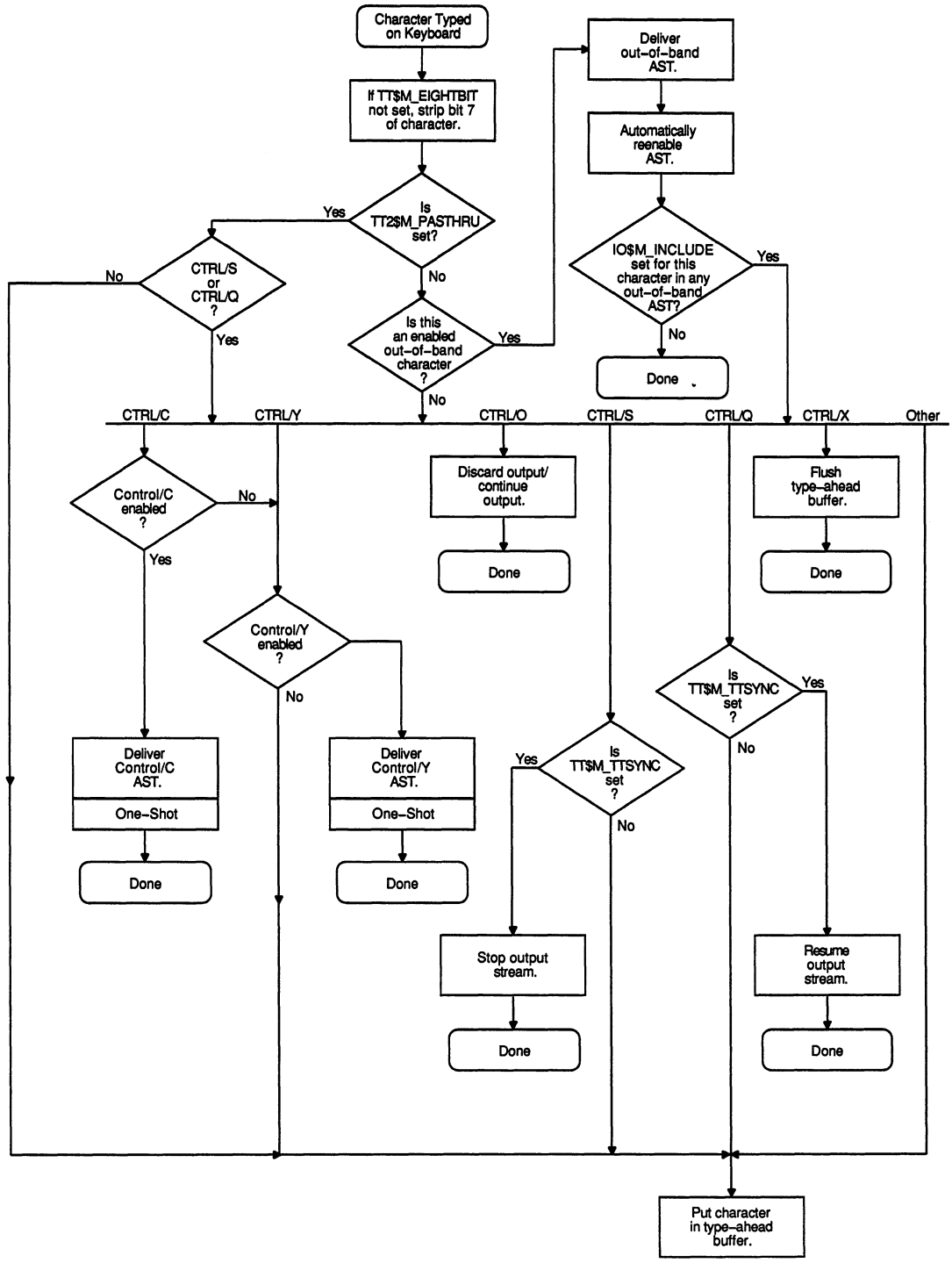
The broadcast function modifier allows you to turn on or turn off selected broadcast requester identifiers (IDs). The following combination of function code and modifier is provided:

- IO$_SETMODE!IO$M_BRDCST

This function code modifier pair takes the following device- or function-dependent arguments:

- P1—A buffer that contains the bits that specify the requester IDs to be broadcast

Figure 8–9 Relationship of Out-of-Band Function with Control Characters



- P2—The length of the P1 buffer (default is eight bytes)

The first longword of P1 is reserved for use by Digital facilities, as shown in Table 8–12. The symbols are defined in the system macro library ($BRKDEF). The second longword is for customer use to specify selected bits. If any bit is set in the P1 buffer, that particular requester ID is turned off for broadcast.

Table 8–12 Broadcast Requester IDs

Bit	Meaning
BRK$C_DCL	Disables broadcasts by CTRL/T
BRK$C_GENERAL	Disables broadcasts by the DCL command REPLY and the SYS$BRDCST system service
BRK$C_MAIL	Disables broadcasts by the Mail Utility
BRK$C_PHONE	Disables broadcasts by the Phone Utility
BRC$C_QUEUE	Disables broadcasts about batch and print queues
BRK$C_SHUTDOWN	Disables broadcasts about system shutdown
BRK$C_URGENT	Disables broadcasts labeled URGENT by the REPLY command
BRK$C_USERn	Disables broadcasts by images associated with the specified value; *n* can be any decimal integer between 1 and 16

8.4.4 LAT Port Driver QIO Interface

The LAT (Local Area Transport) port driver accommodates I/O requests from application programs for connections to remote devices on one or more terminal servers, and I/O requests that support other miscellaneous functions. A remote device, such as a printer, can be shared in a LAT configuration. Before an application program can access a remote device, the VMS system manager must create logical devices on the VMS operating system and map them to physical devices connected to terminal servers. Creating and mapping these logical devices can be done either with the LAT Control Program (LATCP) Utility or with a $QIO request from a program that has PHYS_IO privilege. Once mapped, application programs can establish and terminate connections to these remote devices.

This section describes the QIO interface to the LAT port driver (LTDRIVER) and the functions and function modifiers you use to establish and terminate connections to remote devices. The QIO interface allows application programs to access and modify information contained in the LTDRIVER data structures and to initiate events and obtain status information. You must use these QIO functions to establish a connection to a remote device from an application program. Digital does not support any other methods of connection.

The LTDRIVER responds to TEST SERVICE commands issued at terminal servers that support the TEST SERVICE command, such as the DECserver 200 and DECserver 500.

LAT devices can use all read and write function modifiers listed for the terminal driver function codes except those modifiers that apply to modems (see Sections 8.4.1 and 8.4.2).

The VMS operating system does not support the following set mode or set characteristics function code modifiers for LAT devices:

- IO$M_LOOP
- IO$M_UNLOOP
- TT$M_ALTRPAR
- TT$M_ALTFRAME
- TT$M_MODEM
- TT$M_READSYNC
- TT2$M_SETSPEED

With LAT devices, the terminal server, rather than the VMS host, handles flow control to the physical device. A separate flow control mechanism exists between the server and the host.

8.4.4.1 LAT Port Driver Functions

The VMS operating system provides the following combinations of function code and modifier:

- IO$_TTY_PORT!IO$M_LT_CONNECT—Requests the LAT port driver make a connection to a remote device on a server.
- IO$_TTY_PORT!IO$M_LT_DISCON—Requests the LAT port driver terminate the LAT connection to the remote device.
- IO$_TTY_PORT!IO$M_LT_MAP_PORT—Associates a specific port on a terminal server with a LAT (LTAxxx:) device. Equivalent to the LATCP command SET PORT LTAxxxx:/NODE=server-name /PORT=port-name.
- IO$_TTY_PORT!IO$M_LT_RATING—Sets a static rating for a VMS service. This QIO is equivalent to the LATCP command SET SERVICE/STATIC_RATING=n.

The LAT port driver can only connect to a remote device if it is currently not in use. Table 8–13 lists the conditions that can occur when an application program issues an IO$M_LT_CONNECT request for a connection to a remote device. After a request for a connection is queued on the terminal server, the QIO request is not completed until the connection is established, rejected, or timed out.

Table 8–13 IO$M_LT_CONNECT Request Status

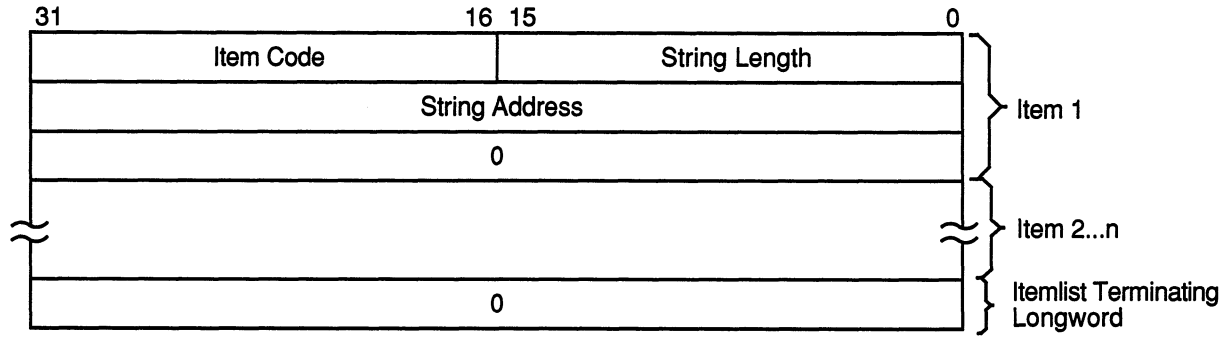
Event	IOSB Status	Explanation
Connection established	SS$_NORMAL	The connection is successful, and the device is ready to use.
Connection timeout	SS$_TIMEOUT	The connection timed out. The server is not available, or an incorrect server name was specified. The timeout period is 5 seconds.
Connection rejected	SS$_ABORT. IOSB+2 contains LAT rejection code.	The connection cannot be made. See Table 8–14 for possible reasons. The LAT port driver updates the I/O status block.
Connection request invalid	No status. SS$_ILLIOFUNC returned in Register 0.	The QIO request is not to an applications port. The LAT port driver rejects the request immediately.
Connection already established on port	No status. SS$_DEVACTIVE returned in Register 0.	The QIO request is for an applications port already in use. The LAT port driver rejects the request immediately.

After you enter a disconnect request (IO$_TTY_PORT!IO$M_LT_DISCON), the applications port's UCB goes off line momentarily. A connect request (IO$_TTY_PORT!IO$M_LT_CONNECT) may return a SS$_DEVACTIVE status if the connect request was immediately preceded by a disconnect request. In this case, reenter the connect request.

The IO$M_LT_MAP_PORT modifier accepts two arguments: P1 and P2. P1 is the address of an item list, which must contain the node name, and either the port name or the service name of the remote terminal server port. (These names must be defined locally on the terminal server.) The item list can also contain the VMS link name and the terminal server Ethernet address. The item list, which must be in type 3 format (see Figure 8–10), is terminated by a longword of 0. The item list contains the following parameters:

- IO$V_LT_MAP_NODNAM—The node name. The node name is the name of the terminal server where the application device is located.
- IO$V_LT_MAP_PORNAM—The port name.
- IO$V_LT_MAP_SRVNAM—The service name.
- IO$V_LT_MAP_LNKNAM—The Ethernet link name, which is always required.
- IO$V_LT_MAP_NETADR—The address of the 6-byte word containing the Ethernet address of the terminal server. IO$V_LT_MAP_NETADR can be substituted for IO$V_LT_MAP_NODNAM.

Figure 8–10 IO$M_LT_MAP_PORT Item List



The P2 argument for IO$M_LT_MAP_PORT is a longword that passes queued status. Bit 0 cleared means nonqueued; bit 0 set means queued.

The IO$M_LT_RATING modifier accepts two arguments: P1 and P2. P1 is the address of the string descriptor that contains the service name, which must already exist. P2 is the rating to assign the service. Ratings range from 0 to 255 (decimal).

Table 8–14 lists the possible status of the I/O Status Block after a IO$M_LT_MAP_PORT or IO$M_LT_RATING request.

Table 8–14 IO$M_LT_MAP_PORT and IO$M_LT_RATING Request Status

Event	Contents of I/O Status Block and R0
Operation successful	SS$_NORMAL
Illegal or incomplete parameter list; non-existent service	SS$_BADPARAM
Access violation in one of the arguments	SS$_ACCVIO
No privilege	SS$_NOPRIV

8.4.4.2 Application Services Creation

Rather than the normal timesharing service offered by the VMS operating system, VMS application programs can make use of LAT application services that allow terminal server users to connect to a specialized application. To do this, the system manager must create LAT ports that are dedicated to a particular application service. When a terminal server user uses the terminal server CONNECT command to connect to an application service, the connection is directly to the VMS application program that controls a LAT port (LTA device) associated with that service. In this case the VMS prompt *Username:* is not received. Digital

recommends that you create application services for VMS service nodes in the following order:

1. Define the dedicated ports in LTLOAD.COM and execute the command procedure in SYSTARTUP_V5.COM. (Refer to the *VMS LAT Control Program (LATCP) Manual* and *Guide to Setting Up a VMS System* for additional information.)
2. Run the application program. Within the application program allocate dedicated ports with the same name as those defined in LTLOAD.COM. Use the Assign I/O Channel ($ASSIGN) system service to assign service channels to the ports.
3. Post a read request to the dedicated ports. When the terminal user connects to the service and presses the Return key, the application program can perform I/O to the dedicated port.
4. To break the connection, use the Deassign I/O Channel ($DASSGN) system service to deassign the channel and the Deallocate Device ($DALLOC) system service to deallocate the device. The application program must reallocate the port and reassign the channel in preparation for the next connection.

An example of the application service concept is a VMS program that provides the time of day. For this example, the system manager includes the following lines in LTLOAD.COM (or enters them manually in the LATCP program):

```
CREATE SERVICE TIME/ID="At the tone, the time will be"
CREATE PORT LTA99:/DEDICATED
SET PORT LTA99:/DEDICATED/SERVICE=TIME
```

An application program then assigns a channel to device LTA99. When a terminal server user types CONNECT TIME, the user is connected to this application program, and the program prints out the time of day. The program then deassigns the channel, which disconnects the server user.

A system manager may associate more than one LAT port with the same service. In that case, the application program that offers the service should assign channels to all of the LTA devices created for that service.

8.4.4.3 Hangup Notification

To allow notification by the terminal driver of abnormal termination during write operations, you should enable a CTRL/Y AST on the channel (see Section 8.4.3.2). This ensures that the terminal driver notifies application programs, which are writing data, of an abnormal connection termination. Note that the VMS operating system does not return an AST parameter to the CTRL/Y AST routine.

When an application program with a pending read request has an abnormal LAT connection termination, the VMS terminal driver returns a SS$_HANGUP status in the first word of the IOSB.

8.4.5 Sense Mode and Sense Characteristics

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

- IO$_SENSEMODE
- IO$_SENSECHAR

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

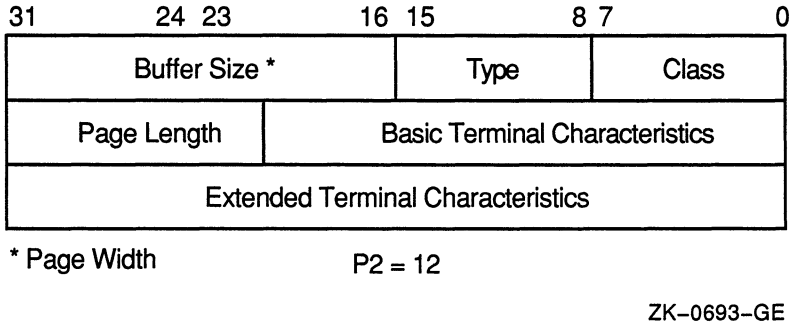
These function codes take the following device- or function-dependent arguments:

- P1—Address of a characteristics buffer
- P2—Length of characteristics buffer (default length is 8 bytes)

  For remote terminals, specify a P2 value of 8 or 12 only.

The P1 argument points to a variable-length block, as shown in Figure 8–11.

Figure 8–11 Sense Mode Characteristics Buffer



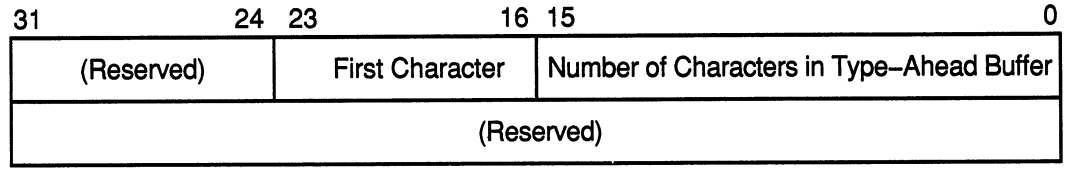
In the buffer, the device class is DC$_TERM, which is defined by the $DCDEF macro. The terminal type is defined by the $TTDEF macro, such as TT$_LA36. The maximum entry for buffer size (page width) is 255. Table 8–5 lists the values for terminal characteristics. Table 8–6 lists the extended terminal characteristics. Characteristics values are defined by the $TTDEF macro.

The sense mode and sense characteristics functions can take the type-ahead count, read modem, and broadcast function modifiers described in the next few sections.

8.4.5.1 Type-ahead Count Function Modifier

The type-ahead count function modifier 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 8–12 shows the format of this buffer.

Figure 8–12 Sense Mode Characteristics Buffer (type-ahead)



ZK–0694–GE

8.4.5.2 Read Modem Function Modifier

The read modem function modifier allows access to controller-dependent information. The following combinations of function code and modifier are provided:

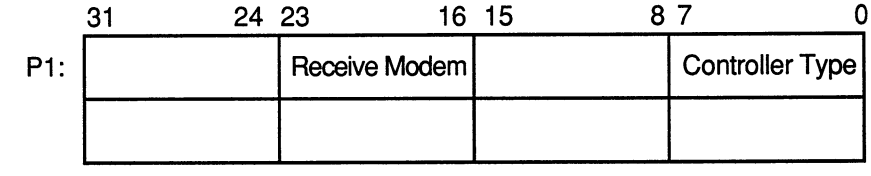
- IO$_SENSEMODE!IO$M_RD_MODEM
- IO$_SENSECHAR!IO$M_RD_MODEM

These function code modifier pairs take the following device- or function-dependent argument:

- P1—The address of a quadword block

Figure 8–13 shows the format of this block.

Figure 8–13 Sense Mode P1 Block



ZK–0695–GE

The receive modem field returns the value of the current input modem signals. Any or all of the following signals can be returned:

- TT$M_DS_DSR—Data set ready (DSR)
- TT$M_DS_RING—Calling indicator (RING)
- TT$M_DS_CARRIER—Data channel received line signal detector (CARRIER)
- TT$M_DS_CTS—Ready for sending (CTS)
- TT$M_DS_SECREC—Received backward channel data (Sec RxD)

The $TTDEF macro defines the symbols for the receive modem field.

The controller type field returns the type of terminal controller in use by the currently active terminal line. The $DCDEF macro defines the symbols for the following types of controllers:

- DT$_DZ11—DZ11 and DZV11
- DT$_DZ32—DZ32
- DT$_DMF32—DMF32
- DT$_DMB32—DMB32
- DT$_DMZ32—DMZ32
- DT$_DHV—DHV11
- DT$_DHU—DHU11
- DT$_LAT—LAT server

Note 1: The IO$M_RD_MODEM function modifier is not supported for LAT devices.

Note 2: The IO$M_RD_MODEM function modifier is not supported for remote terminals. The status SS$_DEVREQERR is returned in the I/O status block.

8.4.5.3 Broadcast Function Modifier

The broadcast function modifier returns those bits that have been set by the set mode function modifier IO$M_BRDCST (see Table 8–12 in Section 8.4.3.6). The following combination of function code and modifier is provided:

- IO$_SENSEMODE!IO$M_BRDCST

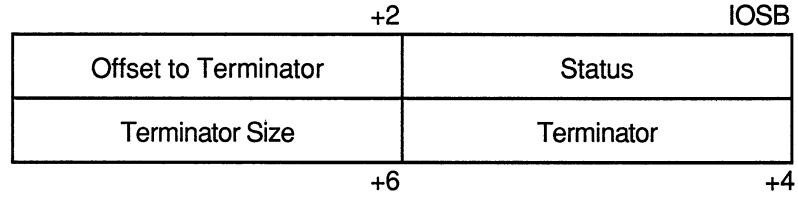
This function code modifier pair takes the following device- or function-dependent arguments:

- P1—A buffer that contains the bits that specify the requester IDs to be broadcast. (If the bit is set in the first longword, that particular command is turned off for broadcast.)
- P2—The length of the P1 buffer.

8.5 I/O Status Block

The I/O status block (IOSB) formats for the read, write, set mode, set characteristics, sense mode, sense characteristics, and LAT port driver I/O functions are shown in Figures 8–14, 8–16, 8–17, and 8–18. Figure 8–15 shows the IOSB format for the itemlist read function. Appendix A lists the status returns for these functions. (The *VMS System Messages and Recovery Procedures Reference Manual* provides explanations and suggested user actions for these returns.)

Figure 8–14 IOSB Contents—Read Function



ZK–0696–GE

Figure 8–15 IOSB Contents—Itemlist Read Function

Offset to Terminator		Status	
Cursor Position from EOL	Terminator Length	(Reserved)	Terminator Character

IOSB Contents: Itemlist Read Function

ZK–1306–GE

Figure 8–16 IOSB Contents—Write Function

Byte Count	Status
0	0

IOSB Contents: Write Function

ZK–1307–GE

Figure 8–17 IOSB Contents—Set Mode, Set Characteristics, Sense Mode, and Sense Characteristics Functions

Receive Speed *	Transmit Speed	Status	
0	Parity Flags	LF Fill Count	CR Fill Count

* Only specified if different than transmit speed.

ZK–0698–GE

In Figure 8–14, the offset to terminator at IOSB+2 is the count of characters before the terminator character (see Section 8.4.1.2). The terminator character 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. In the case of multiple character escape sequences that act as terminators, the terminator at IOSB+4 is the first character (ESC) of the escape sequence. 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 8.2.1.4). The sum of IOSB+2 and IOSB+6 is the number of characters in the buffer.

In Figure 8–15 the terminator position word contains a number, the character of which is determined by the mode of operation. For itemlist read operations that do not specify TRM$K_EM_RDVERIFY, this word contains the number of characters from the end of the buffer to the cursor location at the time the terminator character was received. If TRM$K_EM_RDVERIFY is specified, the terminator position word contains the offset into the buffer from the nonverified character.

The byte at IOSB+5 passes the status information listed in Table 8–15 on TRM$K_EM_RDVERIFY operations in which TRM$M_TM_ARROWS or TRM$M_TM_TOGGLE is set in TRM$_MODIFIERS.

Table 8–15 Byte IOSB+5 Status Information

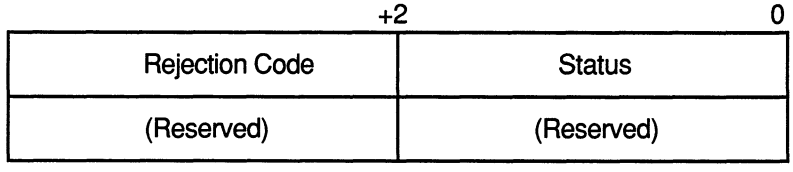
Bit	Interpretation
7 (sign bit)	0 to indicate rest of bits valid. This applies to insert/overstrike and arrow key read verify functionality only.
6-2	Always 0 if bit 7 is equal to 0. Not used; reserved for future use.
1 TRM$V_ST_OTHERWAY	Set to indicate that read is terminated in left-justify insert mode or right-justify overstrike mode.
0 TRM$V_ST_FIELD_FULL	Read terminated on an auto-tab field full condition. IOSB+7 contains an index to the cursor.

In Figure 8–16, the remote terminal driver does not return the number of lines output or the cursor position.

When an application program makes an I/O request for a connection to a remote device on a terminal server, the LAT port driver places status information about the request into the first word of the I/O status block, as shown in Figure 8–18. Tables 8–13 and 8–14 list the possible status returns.

If the server rejects the request, the LAT port driver returns a numeric LAT rejection code in the second word of the I/O status block. Table 8–16 lists the LAT rejection codes.

Figure 8–18 IOSB Contents—LAT Port Driver Function



ZK–6135–GE

Table 8–16 LAT Rejection Codes

Value	Reason
0	Unknown.
2	System shutdown in progress.
5	Insufficient resources at server.
6	Port or service in use.

(continued on next page)

Table 8–16 (Cont.) LAT Rejection Codes

Value	Reason
7	No such service.
8	Service is disabled.
9	Service is not offered on the requested port.
10	Port name is unknown.
13	Immediate access rejected.
14	Access denied.
15	Corrupted request.
16	Requested function is not supported.
17	Session cannot be started.
18	Queue entry deleted by server.
19	Illegal request parameters.

8.6 Terminal Driver Programming Examples

This section contains the following programming examples:

- Example 8–1 shows several I/O operations using the full-duplex capabilities of the terminal.
- Example 8–2 shows a typical read verify operation.
- Example 8–3 shows how to connect to an applications (LT) device.

8.6.1 Terminal I/O Program Example

Example 8–1 illustrates some important concepts about terminal driver programming: assigning an I/O channel, performing full-duplex I/O operations, enabling CTRL/C AST requests, and itemlist read operations. The program is designed to run with a terminal set to full-duplex mode. The initialization code queues a read request to the terminal and enables CTRL/C AST requests. The main loop then prints out a random message every three seconds. When you enter a message on the terminal, the read AST routine prints an acknowledgment message and queues another read request. If you press CTRL/C, the associated AST routine cancels the I/O operation on the assigned channel and exits to the command interpreter.

Example 8–1 Terminal Program Example

```
        .TITLE  FULL_DUPLEX TERMINAL PROGRAMMING EXAMPLE
        .IDENT  /05/
; ********************************************************************
;
;                          TERMINAL PROGRAM
;
; ********************************************************************
        .SBTTL  DECLARATIONS
        .DISABLE        GLOBAL

;
; Declare the external symbols and MACRO libraries.
;
        .EXTERNAL       LIB$GET_EF
        .LIBRARY        'SYS$LIBRARY:LIB.MLB'
        .LIBRARY        'SYS$LIBRARY:STARLET.MLB'
;
; Define symbols
;
        $IODEF                  ; Define I/O function codes
        $QIODEF                 ; Define QIO definition codes
        $SSDEF                  ; Define the system service status codes
        $TRMDEF                 ; Define itemlist read codes
        $TTDEF                  ; Terminal characteristic definitions
;
; Define macros
;
        .SHOW
        .MACRO  ITEM    LEN=0,CODE,VALUE
        .WORD   LEN
        .WORD   TRM$_'CODE'
        .LONG   VALUE
        .LONG   0
        .ENDM   ITEM
        .NOSHOW

;
; Declare exit handler control block
;
EXIT_HANDLER_BLOCK:
        .LONG   0               ; System uses this for pointer
        .LONG   EXIT_HANDLER    ; Address of exit handler
        .LONG   1               ; Argument count for handler
        .LONG   STATUS          ; Destination of status code
STATUS: .BLKL   1               ; Status code from $EXIT

;
; Allocate terminal descriptor and channel number storage
;
TT_DESC:
        .ASCID  /SYS$INPUT/     ; Logical name of terminal
TT_CHAN:
        .BLKW   1               ; TT channel number storage
```

(continued on next page)

Example 8–1 (Cont.) Terminal Program Example

```
;
; Define acknowledgment message.  This is done right above input buffer
; so that we can concatenate the two together when the acknowledgment
; message is issued.
;
ACK_MSG:
        .ASCII  <CR><LF>/Following input acknowledged: /
ACK_MSGLEN=.-ACK_MSG            ; Calculate length of message

;
; Allocate input buffer
;
IN_BUFLEN = 20                  ; Set length of buffer
IN_BUF:
        .BLKB   IN_BUFLEN       ; Allocate character buffer
IN_IOSB:
        .BLKQ   1               ; Input I/O status block
;
; Define out-of-band ast character mask
;
CNTRLA_MASK:
        .LONG   0
        .LONG   ^B0010          ; Control A mask

;
; Define old terminal characteristics buffer
;
OLDCHAR_BUF_LEN = 12
OLDCHAR_BUF:
        .BLKB   OLDCHAR_BUF_LEN

;
; Define new terminal characteristics buffer
;
NEWCHAR_BUF_LEN = 12
NEWCHAR_BUF:
        .BLKB   NEWCHAR_BUF_LEN

;
; Define carriage control symbols
;
        CR=^X0D                 ; Carriage return
        LF=^X0A                 ; Line feed

;
; 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
;
```

(continued on next page)

Example 8–1 (Cont.) Terminal Program Example

```
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$:    .ASCII  <CR><LF>/RED ALERT!!!   RED ALERT!!!/
15$=.-10$
;
20$:    .ASCII  <CR><LF>/ALL SYSTEMS GO/
25$=.-20$
;
30$:    .ASCII  <CR><LF>/WARNING.....INTRUDER ALARM/
35$=.-30$
;
40$:    .ASCII  <CR><LF>/***** SYSTEM OVERLOAD *****/
45$=.-40$
;
; Static QIO packet for message output using QIO$_G form
;
WRITE_QIO:
        $QIO    EFN=SYNC_EFN, - ; QIO packet
                FUNC=IO$_WRITEVBLK!IO$M_BREAKTHRU!IO$M_REFRESH, -
                IOSB=SYNC_IOSB

;
; Declare the required I/O status blocks.
;
SYNC_IOSB::     .BLKQ   1       ; I/O status block for synchronous terminal processing.

;
; Declare the required event flags.
;
ASYNC_EFN::     .BLKL   1       ; Event flag for asynchronous terminal processing.
SYNC_EFN        ==      WRITE_QIO + 4   ; Event flag for sync terminal processing.
TIMER_EFN::     .BLKL   1       ; Event flag for timer processing.

;
; Timer storage
;
WAITIME:
        .LONG   -10*1000*1000*3,-1      ; 3 second delta time
TIME:
        .BLKQ   1               ; Current storage time used for
                                ; random number
```

(continued on next page)

Example 8–1 (Cont.) Terminal Program Example

```
        .PAGE
        .SBTTL  START - MAIN ROUTINE
        .ENABLE LOCAL_BLOCK
;++
;
; Functional description:
;
; ******************************************************************
;
;                       Start program
;
; ******************************************************************
;
;       The following code performs initialization functions.
;       It is assumed that the terminal is already in
;       FULL-DUPLEX mode.
;
;       NOTE: When doing QIO_S calls, parameters P1 and P3-P6 should be
;             passed by value, while P2 should be passed by reference.
;
; Input parameters:
;       None
;
; Output parameters:
;       None
;
;--
        .ENTRY  START   ^M < >

;               Get the required event flags.

        PUSHAL  ASYNC_EFN
        CALLS   # 1, G^ LIB$GET_EF      ; Get EFN for async terminal operations.
        BLBC    R0, 10$                 ; Error - branch.
        PUSHAL  SYNC_EFN
        CALLS   # 1, G^ LIB$GET_EF      ; Get EFN for sync terminal operations.
        BLBC    R0, 10$                 ; Error - branch.
        PUSHAL  TIMER_EFN
        CALLS   # 1, G^ LIB$GET_EF      ; Get EFN for timer operations.
        BLBC    R0, 10$                 ; Error - branch.

;               Initialize the terminal characteristics.

        $ASSIGN_S       DEVNAM=TT_DESC,-; Assign terminal channel using
                        CHAN=TT_CHAN    ; logical name and channel number
        BLBC    R0, 10$                 ; Error - branch.
        BSBW    CHANGE_CHARACTERISTICS  ; Change the characteristics of
                                        ; terminal
        BSBW    ENABLE_CTRLCAST         ; Allow CTRL/C traps
        BSBW    ENABLE_OUTBANDAST       ; Enable CTRL/A out-of-band AST
        BSBW    ENABLE_READ             ; Queue read
        MOVZWL  TT_CHAN, WRITE_QIO+8    ; Insert channel into
        BRB     LOOP                    ; static QIO packet

10$:
        BRW     ERROR
```

(continued on next page)

Example 8–1 (Cont.) Terminal Program Example

```
;
; This loop outputs a message based on a random number and then
; delays for 3 seconds
;
LOOP:
        $GETTIM_S       TIMADR=TIME       ; Get random time
        BLBC    R0, 10$                   ; Error - branch.
        EXTZV   #6, #2, TIME, R0          ; Load random bits into switch
        MOVQ    ARRAY[R0], -              ; Load message address
                WRITE_QIO+QIO$_P1         ; and size into QIO
                                          ; packet

;
; Issue QIO write using packet defined in data area
;
        $QIOW_G WRITE_QIO
        BLBC    R0, 10$                   ; QIO error - branch.
        MOVZWL  SYNC_IOSB, R0             ; Get the terminal driver status.
        BLBC    R0, 10$                   ; Terminal driver error - branch.
;
; Delay for 3 seconds before issuing next message
;
        $SETIMR_S       EFN=TIMER_EFN,- ; Timer service
                        DAYTIM=WAITIME  ; will set event flag
                                        ; in 3 seconds
        BLBC    R0, 10$                 ; Error - branch.
        $WAITFR_S       EFN=TIMER_EFN   ; Wait for event flag
        BLBS    R0, LOOP                ; No error if set
        BRB     10$                     ; Error - branch.

        .DISABLE        LOCAL_BLOCK

        .PAGE
        .SBTTL  CHANGE_CHARACTERISTICS - CHANGE CHARACTERISTICS OF TERMINAL
;++
;
; Functional description:
;
;       Routine to change the characteristics of the terminal.
;
; Input parameters:
;       None
;
; Output parameters:
;       R0 - status from $QIO call.
;       R1 - R5 destroyed
;
;--
;
```

(continued on next page)

Example 8–1 (Cont.) Terminal Program Example

```
CHANGE_CHARACTERISTICS:
        $QIOW_S EFN=SYNC_EFN, -          ; Get current terminal characteristics
                CHAN=TT_CHAN, -
                FUNC=#IO$_SENSEMODE, -
                IOSB=SYNC_IOSB, -
                P1=OLDCHAR_BUF, -
                P2=#OLDCHAR_BUF_LEN
        BLBC    R0, 10$                  ; Error if clear
        MOVZWL  SYNC_IOSB, R0            ; Get the terminal driver status.
        BLBC    R0, 10$                  ; Error - branch

        $DCLEXH_S EXIT_HANDLER_BLOCK     ; Declare exit handler to reset
                                         ; characteristics
        BLBC    R0, 10$                  ; Error - branch.
        MOVC3   #OLDCHAR_BUF_LEN, -      ; Move old characteristics into
                OLDCHAR_BUF, -           ; new characteristics buffer
                NEWCHAR_BUF
        BISL2   #TT$M_NOBRDCST, -        ; Set nobroadcast bit
                NEWCHAR_BUF+4            ; ...
        $QIOW_S EFN=SYNC_EFN, -          ; Set current terminal characteristics
                CHAN=TT_CHAN, -
                FUNC=#IO$_SETMODE, -
                IOSB=SYNC_IOSB, -
                P1=NEWCHAR_BUF, -
                P2=#NEWCHAR_BUF_LEN
        BLBC    R0, 10$                  ; QIO error - branch.
        MOVZWL  SYNC_IOSB, R0            ; Get the terminal driver status.
        BLBC    R0, 10$                  ; Terminal driver error - branch.
        RSB
10$:
        BRW     ERROR

        .PAGE
        .SBTTL  ENABLE_CTRLCAST - ENABLE CTRL/C AST
;++
;
; Functional description:
;
;       Routine to allow CTRL/C recognition.
;
; Input parameters:
;       None
;
; Output parameters:
;       None
;
;--
;
```

(continued on next page)

Example 8–1 (Cont.) Terminal Program Example

```
ENABLE_CTRLCAST:
        $QIOW_S EFN=SYNC_EFN, -
                CHAN=TT_CHAN, -
                FUNC=#IO$_SETMODE!IO$M_CTRLCAST, -
                IOSB=SYNC_IOSB, -
                P1=CTRLCAST, -          ; AST routine address
                P3=#3                   ; User mode
        BLBC    R0, 10$                 ; Error - branch.
        MOVZWL  SYNC_IOSB, R0           ; Get the terminal driver status.
        BLBC    R0, 10$                 ; Terminal driver error - branch.
        RSB

10$:
        BRW     ERROR

        .PAGE
        .SBTTL  ENABLE_OUTBANDAST - ENABLE CTRL/A AST
;++
;
; Functional description:
;
;       Routine to allow CNTRL/A recognition.
;
; Input parameters:
;       None
;
; Output parameters:
;       None
;

ENABLE_OUTBANDAST:
        $QIOW_S EFN=SYNC_EFN, -
                CHAN=TT_CHAN, -
                FUNC=#IO$_SETMODE!IO$M_OUTBAND, -
                IOSB=SYNC_IOSB, -
                P1=CTRLAAST, -          ; AST routine address
                P2=#CNTRLA_MASK, -      ; Character mask
                P3=#3                   ; User mode
        BLBC    R0, 10$                 ; QIO error - branch.
        MOVZWL  SYNC_IOSB, R0           ; Get the terminal driver status.
        BLBC    R0, 10$                 ; Terminal driver error - branch.
        RSB

10$:
        BRW     ERROR
```

(continued on next page)

Example 8–1 (Cont.) Terminal Program Example

```
        .PAGE
        .SBTTL  ENABLE_READ - QUEUE A READ TO THE TERMINAL.
;++
;
; Functional description:
;
;       Routine to queue a read operation to the terminal.
;
; Input parameters:
;       None
;
; Output parameters:
;       None
;
; Define item list for itemlist read
;
ITEM_LST:
        ITEM    0, MODIFIERS, -                 ; Convert lowercase to
                TRM$M_TM_CVTLOW!TRM$M_TM_NOEDIT ; upper and inhibit line
        ITEM    6, TERM,MASK_ADDR               ; editing
                                                ; Set up terminator mask

ITEM_LEN        =       . - ITEM_LST
MASK_ADDR:
        .LONG   1@^XD                   ; Terminator mask is <CR>
        .WORD   1@4                     ; and "$"
ENABLE_READ:
        $QIO_S  EFN=ASYNC_EFN, -        ; Must not be QIOW form or read will block
                CHAN=TT_CHAN, -         ; process
                FUNC=#IO$_READVBLK!IO$M_EXTEND, -
                IOSB=IN_IOSB, -
                ASTADR=READAST, -       ; AST routine to execute
                P1=IN_BUF, -            ; on
                P2=#IN_BUFLEN, -
                P5=#ITEM_LST, -         ; Itemlist read address
                P6=#ITEM_LEN            ; Itemlist read size
        BLBC    R0, 10$                 ; QIO error - branch.

; The queued read operation will not affect write operations due
; to the fact that breakthru has been set for the write operations.

        RSB

10$:
        BRW     ERROR
```

(continued on next page)

Example 8–1 (Cont.) Terminal Program Example

```
        .PAGE
        .SBTTL  READAST - AST ROUTINE FOR READ COMPLETION
        .ENABLE LOCAL_BLOCK
;++
;
; Functional description:
;
;       AST routine to execute on read completion.
;
; Input parameters:
;       None
;
; Output parameters:
;       None
;
;--
;
10$:
        MOVZWL  IN_IOSB, R0             ; Get the terminal driver status
20$:
        BRW     ERROR                   ; Exit with error status.

        .ENTRY  READAST         ^M < R2, R3, R4, R5 >   ; Procedure entry mask
        BLBC    IN_IOSB, 10$            ; Terminal driver error - branch
        MOVZWL  IN_IOSB+2, R0           ; Get number of characters read into R0
        ADDL2   #ACK_MSGLEN, R0         ; Add size of fixed acknowledge message
        $QIO_S  EFN=ASYNC_EFN, -        ; Issue acknowledge message
                CHAN=TT_CHAN, -         ; Note, ACK must be asynchronous (QIO)
                FUNC=#IO$_WRITEVBLK, -  ; and the terminal driver write status
                P1=ACK_MSG, -           ; is ignored (no IOSB and AST routine).
                P2=R0                   ; Specify IOSB and AST routine if output
                                        ; must be displayed on the terminal.
        BLBC    R0, 20$                 ; QIO error - branch
;
; Process read message
;
;        .
;        .
;        .
;(user-provided code to decode command inserted here)
;        .
;        .
;        .
        BSBW    ENABLE_READ             ; Queue next read
        RET                             ; Return to mainline loop

        .DISABLE        LOCAL_BLOCK

        .PAGE
        .SBTTL  CTRLAAST - AST ROUTINE FOR CTRL/A
        .SBTTL  CTRLCAST - AST ROUTINE FOR CTRL/C
        .SBTTL  ERROR - EXIT ROUTINE
```

(continued on next page)

Example 8–1 (Cont.) Terminal Program Example

```
;++
;
; Functional description:
;
;       AST routine to execute when CTRL/C or CTRL/A is entered.
;
; Input parameters:
;       None
;
; Output parameters:
;       None
;

CTRLCAST::
CTRLAAST::
        .WORD   ^M < >                  ; Procedure entry mask
        MOVL    #SS$_NORMAL, R0         ; Put success in R0

ERROR::
        $EXIT_S R0                      ; Exit
        RSB

        .PAGE
        .SBTTL  EXIT_HANDLER - EXIT HANDLER ROUTINE
;++
;
; Functional description:
;
;       Exit handler routine to execute when image exits.  It cancels
;       any outstanding I/O on this channel and resets the terminal
;       characteristics to their original state.
;
; Input parameters:
;       None
;
; Output parameters:
;       None
;
;--
;

        .ENTRY  EXIT_HANDLER    ^M< >
        $CANCEL_S       CHAN=TT_CHAN    ; Flush any I/O on queue
        $QIOW_S EFN=SYNC_EFN, -         ; Reset terminal characteristics
                CHAN=TT_CHAN, -
                FUNC=#IO$_SETMODE, -
                IOSB=SYNC_IOSB, -
                P1=OLDCHAR_BUF, -
                P2=#OLDCHAR_BUF_LEN
        BLBC    R0, 10$                 ; QIO error - branch.
        MOVZWL  SYNC_IOSB, R0           ; Get the terminal driver status.

10$:
        RET

        .END     START
```

8.6.2 Read Verify Program Example

Example 8–2 is an example of the read verify function. The program shows a typical build of itemlists (both the right and left fields), channel assignment, a right- and left-justified read verify operation, and then the read QIO operation.

Example 8–2 Read Verify Program Example

```
        .TITLE READ_VERIFY - Read Verify Coding Example
        .IDENT  'V05-000'

        .SBTTL  DECLARATIONS
        .DISABLE        GLOBAL

;
; Declare the external system routines and MACRO libraries.
;
        .EXTERNAL       LIB$GET_EF
        .EXTERNAL       SCR$ERASE_PAGE

        .LIBRARY        'SYS$LIBRARY:LIB.MLB'
        .LIBRARY        'SYS$LIBRARY:STARLET.MLB'
;
; Include files:
;
        $IODEF
        $TRMDEF
;
; Macros:
;
.MACRO ITEM LEN=0,CODE,VALUE
        .WORD   LEN
        .WORD   TRM$_'CODE'
        .LONG   VALUE
        .LONG   0
.ENDM ITEM

;
; Equated symbols:
;
INBUF_LEN = 20
ESC = ^X1B

;
; Own storage:
;
; Build item lists for the read verify QIO
;

;
; Right-justified field
;
R_ITEM_LIST:
        ITEM    CODE    = MODIFIERS, -
                VALUE   = TRM$M_TM_R_JUST       ; Right justify

        ITEM    CODE    = EDITMODE, -
                VALUE   = TRM$K_EM_RDVERIFY     ; Enable read verify
```

(continued on next page)

Example 8–2 (Cont.) Read Verify Program Example

```
        ITEM    CODE    = PROMPT, -
                VALUE   = R_PROMPT_ADDR, -
                LEN     = R_PROMPT_LEN          ; Set up prompt

        ITEM    CODE    = INISTRNG, -
                VALUE   = R_INISTR_ADDR, -
                LEN     = R_INISTR_LEN          ; Set up initial string

        ITEM    CODE    = INIOFFSET, -
                VALUE   = R_INISTR_LEN

        ITEM    CODE    = PICSTRNG, -
                VALUE   = R_PICSTR_ADDR, -
                LEN     = R_PICSTR_LEN          ; Set up picture string

        ITEM    CODE    = FILLCHR, -
                VALUE   = <^A/* />            ; clear = *, fill = space

R_ITEM_LIST_LEN = .-R_ITEM_LIST

R_PROMPT_ADDR:
        .ASCII  <ESC>/[12;12H$/
R_PROMPT_LEN = .-R_PROMPT_ADDR

R_INISTR_ADDR:
        .ASCII  /   ,   /
R_INISTR_LEN = .-R_INISTR_ADDR

MASK = TRM$M_CV_NUMERIC!TRM$M_CV_NUMPUNC

R_PICSTR_ADDR:
        .BYTE   MASK
        .BYTE   MASK
        .BYTE   MASK
        .BYTE   0               ; Marker character
        .BYTE   MASK
        .BYTE   MASK
        .BYTE   MASK
R_PICSTR_LEN = .-R_PICSTR_ADDR
;
; Left-justified field
;
L_ITEM_LIST:
        ITEM    CODE    = MODIFIERS, -
                VALUE   = TRM$M_TM_CVTLOW!TRM$M_TM_AUTO_TAB
                                        ; Upcase input and
                                        ; complete on field full

        ITEM    CODE    = EDITMODE, -
                VALUE   = TRM$K_EM_RDVERIFY     ; Enable read verify

        ITEM    CODE    = PROMPT, -
                VALUE   = L_PROMPT_ADDR, -
                LEN     = L_PROMPT_LEN          ; Set up prompt

        ITEM    CODE    = INISTRNG, -
                VALUE   = L_INISTR_ADDR, -
                LEN     = L_INISTR_LEN          ; Set up initial string

        ITEM    CODE    = INIOFFSET, -
                VALUE   = 0
```

(continued on next page)

Example 8–2 (Cont.) Read Verify Program Example

```
        ITEM    CODE    = PICSTRNG, -
                VALUE   = L_PICSTR_ADDR, -
                LEN     = L_PICSTR_LEN          ; Set up picture string

        ITEM    CODE    = FILLCHR, -
                VALUE   = <^A/* />              ; clear = *, fill = space

L_ITEM_LIST_LEN = .-L_ITEM_LIST

L_PROMPT_ADDR:
        .ASCII  <ESC>/[13;12H Enter Date: /
L_PROMPT_LEN = .-L_PROMPT_ADDR

L_INISTR_ADDR:
        .ASCII  /  -   -  /
L_INISTR_LEN = .-L_INISTR_ADDR

MASK1 = TRM$M_CV_NUMERIC
MASK2 = TRM$M_CV_UPPER!TRM$M_CV_LOWER

L_PICSTR_ADDR:
        .BYTE   MASK1
        .BYTE   MASK1
        .BYTE   0               ; Marker character
        .BYTE   MASK2
        .BYTE   MASK2
        .BYTE   MASK2
        .BYTE   0               ; marker character
        .BYTE   MASK1
        .BYTE   MASK1
L_PICSTR_LEN = .-L_PICSTR_ADDR

IN_IOSB:        .BLKL   2
TT_CHAN:        .BLKW   1
INBUF:          .BLKB   INBUF_LEN
SYSINPUT:       .ASCID  /SYS$INPUT/
SYNC_EFN:       .BLKL   1

        .PAGE

        .ENTRY  READ_VERIFY     ^M < >

;
; Get the required event flags.
;

        PUSHAL  SYNC_EFN
        CALLS   # 1, G^ LIB$GET_EF
        BLBC    R0, ERROR                       ; Error - branch

;
; Assign the channel to SYS$INPUT
;

        $ASSIGN_S -
                CHAN = TT_CHAN -
                DEVNAM = SYSINPUT               ; SYS$INPUT
        BLBC    R0, ERROR                       ; Branch on error

;
; Clear the screen
;
```

(continued on next page)

Example 8–2 (Cont.) Read Verify Program Example

```
        CLRQ    -(SP)
        CALLS   #2, G^ SCR$ERASE_PAGE
        BLBC    R0, ERROR

;
; Do the right-justified read operation
;

        PUSHL   #R_ITEM_LIST_LEN
        PUSHAB  R_ITEM_LIST
        CALLS   #2, DO_READ
        BLBC    R0, ERROR

;
; Do the left-justified read operation
;

        PUSHL   #L_ITEM_LIST_LEN
        PUSHAB  L_ITEM_LIST
        CALLS   #2, DO_READ
        BLBC    R0, ERROR

ERROR:
        RET

        .PAGE
;++
;
; DO_READ - do the actual QIO
;
; Inputs:
;
;       4(AP)   the address of the itemlist
;       8(AP)   the length of the itemlist
;
;--

        .ENTRY  DO_READ, ^M<>

        $QIOW_S EFN=SYNC_EFN, -
                CHAN = TT_CHAN, -
                FUNC = #<IO$_READVBLK!IO$M_EXTEND>, -
                IOSB = IN_IOSB, -
                p1 = inbuf, -
                p2 = #inbuf_len, -
                p5 = 4(AP), -
                P6 = 8(AP)
        BLBC    R0, 10$                 ; QIO error - branch
        MOVZWL  IN_IOSB, R0             ; Get the terminal driver status.
        BLBC    R0, 10$                 ; Terminal driver error - branch

;  Handle the input...

10$:
        RET

        .END READ_VERIFY
```

8.6.3 LAT Application Device Program Example

Example 8–3 requests a connection to an applications (LT) device. The program uses the terminal port function code (IO$_TTY_PORT) and the function code modifiers for the LAT port driver to solicit the connection to the applications device. (Note that the IO$_TTY_PORT function is not specific to LAT operations.)

Example 8–3 also illustrates the use of the set rating (IO$M_LT_RATING) and map port (IO$M_LT_MAP_PORT) functions (see Section 8.4.4.1).

See Section 8.4.4.2 for additional information on LAT application programming.

Example 8–3 LAT Application Device Program

```
        .TITLE    LAT APPLICATION DEVICE PROGRAMMING EXAMPLE
        .IDENT    /1.3/

;*************************************************************
;
;               LAT Application Device Program
;
;*************************************************************

        .SBTTL   DECLARATIONS
        .DISABLE        GLOBAL

;
; Declare the external system routines and libraries.
;
        .EXTERNAL       LIB$GET_EF

        .LIBRARY        'SYS$LIBRARY:LIB.MLB'
        .LIBRARY        'SYS$LIBRARY:STARLET.MLB'

;
; Define symbols
;
        $IODEF                          ; I/O function codes
        $QIODEF                         ; QIO definition codes
        $SSDEF                          ; System Service completion codes

;
; Declare the required event flags
;
SYNC_EFN::      .BLKL   1
ASYNC_EFN::     .BLKL   1

;
; Declare exit handler control block
;
EXIT_HANDLER_BLOCK:
        .LONG    0                      ; System uses this for pointer
        .LONG    EXIT_HANDLER           ; Address of exit handler
        .LONG    1                      ; Argument count for handler
        .LONG    STATUS                 ; Destination of status code
STATUS:          .BLKL   1              ; Status code from $EXIT
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
;
; Allocate terminal descriptor and channel number storage
;

TT_DESC:        .ASCID  /SYS$INPUT/     ; Name of terminal
TT_CHAN:        .BLKW   1               ; TT channel number storage
LT_DESC:        .ASCID  /LAT$PORT/      ; Logical name of LT device,
                                        ;  define this to be the application
                                        ;  port created in LATCP
LT_CHAN:        .BLKW   1               ; LT channel number storage

;
; Define carriage control symbols
;
        CR=^X0D                          ; Carriage return
        LF=^X0A                          ; Line feed

;
; Define I/O buffer sizes
;

IN_BUFLEN = 80                          ; Input buffer size
OUT_MSGLEN = 2                          ; Initial length of output message

;
; Allocate I/O buffers; OUT_MSG must appear before IN_BUF.  The effect of this
; is to prefix <CR><LF> to the bytes read from the TT device.  By adding two
; to the number of bytes read, and using the buffer starting at OUT_MSG, the
; message written to the LTA device will be the message read from the TT device
; prefixed with <CR><LF>.
;

OUT_MSG:        .ASCII  <CR><LF>        ; Start address of output buffer
IN_BUF:         .BLKB   IN_BUFLEN       ; Allocate character input buffer

;
; Allocate I/O status blocks (IOSBs)
;

IN_IOSB:        .BLKQ   1               ; Input I/O status block (IOSB)
OUT_IOSB:       .BLKQ   1               ; Output I/O status block (IOSB)
SOL_IOSB:       .BLKQ   1               ; Solicitation connect IOSB
MAP_IOSB:       .BLKQ   1               ; Map IOSB
RATING_IOSB:    .BLKQ   1               ; Rating IOSB
SYNC_IOSB:      .BLKQ   1               ; IOSB for synchronous operations.
ASYNC_IOSB:     .BLKQ   1               ; IOSB for asynchronous operations.
SERVICE_DESC:   .LONG   SERVICE_NAME_LENGTH
                .ADDRESS  SERVICE_NAME

SERVICE_NAME:   .ASCII  /TIMESHARING/   ; Service that was created by LATCP
SERVICE_NAME_LENGTH = . - SERVICE_NAME
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
NEW_RATING:     .LONG   100             ; The new static rating value for it

LATNOD: .ASCII   /PLUTO/

NODLEN= .-LATNOD

LATPORT: .ASCII /APPLIC_DEVICE/

PORTLEN= .-LATPORT
LINKNAM: .ASCII /LAT$LINK/

LINKLEN= .-LINKNAM

MAP_ITEMLIST:
         .WORD   NODLEN
         .WORD   IO$V_LT_MAP_NODNAM
         .ADDRESS LATNOD
         .LONG   0

         .WORD   PORTLEN
         .WORD   IO$V_LT_MAP_PORNAM
         .ADDRESS LATPORT
         .LONG   0

         .WORD   LINKLEN
         .WORD   IO$V_LT_MAP_LNKNAM
         .ADDRESS LINKNAM
         .LONG   0

         .LONG   0

;
; Define output messages
;
; Messages are accessed by indexing into a table of longwords
; with each message described by a message address and length.
; Although not done here, this table should be large enough
; to accomodate all possible reject reasons.
;
MSG_TABLE:                              ; Table of message address
                                        ; and length
        .LONG    01$                    ; First message address
        .LONG    05$                    ; First message length
        .LONG    10$                    ; Message address
        .LONG    15$                    ; Message length
        .LONG    20$                    ; Message address
        .LONG    25$                    ; Message length
        .LONG    30$                    ; Message address
        .LONG    35$                    ; Message length
        .LONG    40$                    ; Message address
        .LONG    45$                    ; Message length
        .LONG    50$                    ; Message address
        .LONG    55$                    ; Message length
        .LONG    60$                    ; Message address
        .LONG    65$                    ; Message length
        .LONG    190$                   ; Message address
        .LONG    195$                   ; Message length
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
;
; Messages  (Refer to the list of LAT rejection codes in Table 8-16.)
;
01$:    .ASCII   /REASON UNKNOWN/
05$=.-01$
;
10$:    .ASCII   <CR><LF>/CONNECTION ESTABLISHED/
15$=.-10$
;
20$:    .ASCII   /SYSTEM SHUTDOWN IN PROGRESS/
25$=.-20$
;
30$:    .ASCII   /REASON UNKNOWN/
35$=.-30$
;
40$:    .ASCII   /REASON UNKNOWN/
45$=.-40$
;
50$:    .ASCII   /INSUFFICIENT RESOURCES/
55$=.-50$
;
60$:    .ASCII   /PORT OR SERVICE IN USE/
65$=.-60$
190$:   .ASCII   /ILLEGAL REQUEST PARAMETERS/
195$=.-190$
;
NOTCON: .ASCII   <CR><LF>/CONNECTION REJECTED - /
NOTCONL=.-NOTCON
;
; Static QIO packets for message output using QIO$_G form

;
WRITE_QIO:
        $QIO     FUNC=IO$_WRITEVBLK!IO$M_BREAKTHRU!IO$M_REFRESH,-
                 EFN=1
;
ERROR_QIO:
        $QIO     FUNC=IO$_WRITEVBLK!IO$M_BREAKTHRU!IO$M_REFRESH,-
                 EFN=1

;
;
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
        .PAGE
        .SBTTL   MAIN ROUTINE
;++
;
; Functional description:
;
;*************************************************************
;
;                    Main Program Routine
;
;*************************************************************
;
;   The following code assigns a channel to the LTAxxx:
;   application device and attempts to create a connection to
;   that device.  The connection status is displayed on
;   the user's terminal.  Input from the user's terminal
;   is output on the LTAxxx device:  CTRL/C entered by the
;   user terminates the program.
;
; Input parameters:
;   None
;
; output parameters:
;   None
;
;--
        .ENTRY  START   ^M <>           ; Entry mask

;
; Get the required event flags.
;
        PUSHAL      ASYNC_EFN
        CALLS       #1, G^LIB$GET_EF
        BLBC        R0,20$              ; Error - branch
        PUSHAL      SYNC_EFN
        CALLS       #1, G^LIB$GET_EF
        BLBC        R0,20$              ; Error - branch

;
; Assign channels
;
        $ASSIGN_S   DEVNAM=TT_DESC, -   ; Assign channel to user's
                    CHAN=TT_CHAN        ; terminal
        BLBC        R0, 20$             ; Error - branch
        $ASSIGN_S   DEVNAM=LT_DESC, -   ; Assign channel to LT device
                    CHAN=LT_CHAN
        BLBC        R0, 20$             ; Error - branch

;
; Declare an exit handler which will execute at image exit.
;
        $DCLEXH_S   DESBLK=EXIT_HANDLER_BLOCK
        BLBC        R0, 20$             ; Error - branch
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
;
;   Perform the privileged operations of changing a static service
;   rating (an operation unrelated to the use of application devices but
;   included here strictly as an example) and remapping the applications
;   port.
;
        BSBW        LAT_SET_RATING          ; Change a static service rating
        BLBC        R0, 20$                 ; Error - branch
        BSBW        LAT_MAP_PORT            ; Change the application port mapping
        BLBC        R0, 20$                 ; Error - branch

;
;   Enable CTRL/C on user terminal and CTRL/Y on LTA device.  Solicit
;   connection to application device and post read to user's terminal.
;   The CTRL/Y AST enable is done in the solicit connection AST so
;   that a CTRL/Y is not returned before the solicit AST.  This
;   prevents the rejection errors from being displayed.
;
        BSBW        ENABLE_CTRLCAST         ; Enable CTRL/C ASTs
        BLBC        R0, 20$                 ; Error - branch
        BSBW        SOL_CONNECT             ; Try to connect to LT device
        BLBC        R0, 20$                 ; Error - branch

10$:
        $HIBER_S                            ; Wait for a while.
        BLBS        R0, 10$                 ; Keep looping until CTRL/C

20$:
        BRW     ERROR

        .PAGE
        .SBTTL      ENABLE_CTRLYAST - Enable CTRLYAST on LTAxxx device
;++
; Functional description:
;
;   Routine to allow hangup notification.  This routine enables
;   CTRL/Y AST delivery for the LTAxxx: device.  The CTRL/Y AST
;   is called if an abnormal termination occurs to the remote
;   application device.
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   R0 = QIO status.
;
;--
ENABLE_CTRLYAST:
        $QIO_S CHAN=LT_CHAN,-
                FUNC=#IO$_SETMODE!IO$M_CTRLYAST,-
                IOSB=ASYNC_IOSB,-
                ASTADR=START_READ,-         ; Start Read at completion of QIO
                P1=HANGUP,-                 ; AST routine address
                P3=#3                       ; User mode
        BLBC    R0, 10$                     ; QIO error - branch
        RSB
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
10$:
        BRW     ERROR

        .PAGE
        .SBTTL          HANGUP - AST Routine for CTRL/Y
;++
;
; Functional description:
;
;   AST routine to execute when CTRL/Y status is returned for the
;   application device.  This status is returned when the
;   connection to the remote device is abnormally terminated.
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   None
;
;--
;
HANGUP:
        .WORD           ^M<>
        MOVZWL          #SS$_HANGUP,R0              ; Indicate hangup
        BRW             ERROR                       ; and exit

        .PAGE
        .SBTTL          START_READ - Start reads on the TT device
;++
; Functional description:
;
;   This routine executes at completion of the SETMODE QIO to set the CTRL/Y
;   AST.  It checks the completion status of the QIO, and starts reads on the
;   TT device by calling ENABLE_READ.
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   None
;
;--
;
START_READ:
        .WORD   ^M<>
        BLBC    ASYNC_IOSB,10$              ; Terminal driver error - branch
        BSBW    ENABLE_READ
        RET
10$:
        MOVZWL  ASYNC_IOSB, R0              ; Put error status in R0
        BRW     ERROR
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
        .PAGE
        .SBTTL        ENABLE_READ - QUEUE A READ TO THE TERMINAL
;++
;
; Functional description:
;
;   Routine to queue a read to the terminal.  The queued
;   read will not affect writes due to the fact that
;   breakthru has been set for writes.
;
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   R0 = Successful QIO status.
;
;--
;
ENABLE_READ:
        $QIO_S  EFN=ASYNC_EFN, -        ; Must not be QIOW form
                CHAN=TT_CHAN, -
                FUNC=#IO$_READVBLK, -
                IOSB=IN_IOSB, -
                ASTADR=READAST, -
                P1=IN_BUF, -
                P2=#IN_BUFLEN
        BLBC    R0, 10$                 ; QIO error - branch
        RSB

10$:
        BRW     ERROR

        .PAGE
        .SBTTL   READAST - AST Routine for Read Completion
;++
;
; Functional description:
;
;   AST routine to execute on read completion.  The data that
;   was input from the users terminal is output on the
;   application device.  Another read request is then posted.
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   None
;
;--
;
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
        .ENTRY  READAST ^M < R2, R3, R4, R5 >   ; Procedure entry mask
        BLBC    IN_IOSB, 10$            ; Terminal driver error - branch
        MOVZWL  IN_IOSB+2, R0           ; Get number of characters read
        ADDL2   #OUT_MSGLEN, R0         ; Add size of fixed ack
        $QIO_S  EFN=ASYNC_EFN, -        ; Output message to LT device
                CHAN=LT_CHAN, -         ; Must be asynchronous (QIO)
                FUNC=#IO$_WRITEVBLK, -
                IOSB=OUT_IOSB, -
                ASTADR=WRITEAST, -
                P1=OUT_MSG, -
                P2=R0
        BLBC    R0, 20$                 ; QIO error - branch
        BSBW    ENABLE_READ             ; Queue next read
        RET

10$:
        MOVZWL  IN_IOSB, R0             ; Put error status in R0

20$:
        BRW     ERROR                   ; Exit with error

        .PAGE
        .SBTTL   WRITEAST - AST Routine for Write Completion
;++
;
; Functional description:
;
;   AST routine to execute on write completion.  Check the status
;   of the write.
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   None
;
;--
;
        .ENTRY WRITEAST ^M < R2, R3, R4, R5 >   ; Procedure entry mask
        BLBC    OUT_IOSB, 10$           ; Terminal driver error - branch
        RET
10$:
        MOVZWL  OUT_IOSB, R0            ; Put error status in R0
        BRW     ERROR
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
        .PAGE
        .SBTTL   ENABLE_CTRLCAST - ENABLE CTRL/C AST
;++
; Functional description:
;
;   Routine to allow CTRL/C recognition on user's terminal
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   R0 = QIO/Terminal driver status.
;
;--
ENABLE_CTRLCAST:
        $QIOW_S EFN=SYNC_EFN, -
                CHAN=TT_CHAN, -
                FUNC=#IO$_SETMODE!IO$M_CTRLCAST, -
                IOSB=SYNC_IOSB, -
                P1=CTRLCAST, -          ; AST routine address
                P3=#3                   ; User mode
        BLBC    R0, 10$                 ; QIO error - branch
        MOVZWL  SYNC_IOSB, R0           ; Get the terminal driver status.
10$:
        RSB

        .PAGE
        .SBTTL   CTRLCAST - AST Routine for CTRL/C
;++
;
; Functional description:
;
;   AST routine to execute when CTRL/C is received.  The connection
;   to the application device is stopped and the program is terminated
;   with normal completion status.
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   None
;
;--
;
CTRLCAST:
        .WORD   ^M<>
        $QIO_S  EFN=ASYNC_EFN, -        ; Disconnect session to LT device
                CHAN=LT_CHAN, -
                FUNC=#IO$_TTY_PORT!IO$M_LT_DISCON, -
                IOSB = ASYNC_IOSB,-
                ASTADR = DISCONNECTAST
        BLBC    R0, 10$                 ; QIO error - branch
        RET
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
10$:
        BRW     ERROR

        .PAGE
        .SBTTL  DISCONNECTAST - AST Routine for disconnect status
;++
;
; Functional description
;
;   AST routine to execute when disconnect is complete.  The image
;   will exit with the status from the disconnect QIO by moving the
;   value in the status field of the IOSB into R0.
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   None
;
;--
;
DISCONNECTAST:
        .WORD   ^M<>
        MOVZWL  ASYNC_IOSB, R0          ; Save disconnect status
ERROR:
        $EXIT_S CODE=R0                 ; Exit
        RET
        .PAGE
        .SBTTL   SOL_CONNECT - Solicit Connection to LT Device
;++
;
; Functional description:
;
;   This routine issues the QIO to the LT driver to solicit
;   the connection to the LT device.
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   R0 = QIO status.
;
;--
;
SOL_CONNECT:
        $QIO_S  EFN=SYNC_EFN, -
                CHAN=LT_CHAN, -
                FUNC=#IO$_TTY_PORT!IO$M_LT_CONNECT, -
                IOSB=SOL_IOSB, -
                ASTADR=SOLAST
        RSB

        .PAGE
        .SBTTL   SOLAST - AST Routine for connection solicitation status
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
;++
;
; Functional description:
;
;   AST routine to execute when connection solicitation is
;   complete.  If status is success, print success message and
;   return.  If status is rejection, print reject message,
;   reject reason, and exit.  If status is otherwise, exit.
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   None
;
;--
;
SOLAST:
        .WORD     ^M<>
        MOVZWL    SOL_IOSB,R0               ; Get return status
        BLBC      R0,10$                    ; If clear, error
        MOVL      R0,R1                     ; Copy success code for index
        JSB       WRITE_STATUS              ; Output success message
        BSBW      ENABLE_CTRLYAST           ; Enable CTRL/Y ASTs
        RET

10$:    CMPW      R0,#SS$_ABORT             ; Is this a rejected connection?
        BNEQ      ERROR                     ; If eq, output error message
        MOVZWL    TT_CHAN,ERROR_QIO+8       ; Insert channel into QIO packet
        $QIOW_G   ERROR_QIO                 ; Output error message first
        MOVZWL    SOL_IOSB+2,R1             ; Set R1 for offset into table
        BSBB      WRITE_STATUS              ; Output error reason
        BRW       ERROR                     ; Exit

WRITE_STATUS:
        MOVQ      MSG_TABLE[R1],-           ; Put message into QIO
                  WRITE_QIO+QIO$_P1
        MOVZWL    TT_CHAN,WRITE_QIO+8       ; Insert channel into QIO packet
        $QIOW_G   WRITE_QIO
        RSB
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
        .PAGE
        .SBTTL   EXIT_HANDLER:
;++
;
; Functional description:
;
;   Exit handler routine to execute when image exits.  It will
;   cancel any outstanding I/O on these channels.
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   None
;
;--
;
EXIT_HANDLER:
        .WORD
        $CANCEL_S  CHAN=TT_CHAN          ; Flush any output
        $CANCEL_S  CHAN=LT_CHAN
        RET

        .PAGE
        .SBTTL     LAT_MAP_PORT  Map Port QIO
;++
;
; Functional description:
;
;   The following code performs a map port QIO. It uses an item
;   list that specifies the names for the target server and port
;   that is associated with the application port.
;
; Input parameters:
;
;   None
;
; Output parameters:
;
;   R0 = QIO/Terminal driver status.
;
;--
LAT_MAP_PORT:
        $QIOW_S EFN = SYNC_EFN, -
                CHAN = LT_CHAN, -
                FUNC = #<IO$_TTY_PORT!IO$M_LT_MAP_PORT>, -
                IOSB = MAP_IOSB, -
                P1 =   MAP_ITEMLIST
        BLBC    R0, 10$
        MOVZWL  MAP_IOSB, R0
```

(continued on next page)

Example 8–3 (Cont.) LAT Application Device Program

```
10$:
        RSB

        .PAGE
        .SBTTL   LAT_SET_RATING
;++
;
; Functional description:
;
;   The following code performs the set rating QIO.
;   In this example, the rating for the service "TIMESHARING"
;   is set to a static rating of 100.
;
; Input parameters:
;   None
;
; Output parameters:
;   R0 = QIO/Terminal driver status.
;
;--
LAT_SET_RATING:

        $QIOW_S EFN = SYNC_EFN, -
                CHAN = LT_CHAN, -
                FUNC = #<IO$_TTY_PORT!IO$M_LT_RATING>, -
                IOSB = RATING_IOSB,-
                P1 =   SERVICE_DESC,-
                P2 =   NEW_RATING
        BLBC    R0, 10$                 ; Error - branch
        MOVZWL  RATING_IOSB, R0

10$:
        RSB

        .END START
```

9 Pseudoterminal Driver

This chapter describes the use of the VMS pseudoterminal driver (FTDRIVER) and the VMS pseudoterminal software.

A pseudoterminal is a software device that appears as a real terminal to an application communicating with it but that does not require the existence of a physical terminal. A pseudoterminal consists of two components: the pseudoterminal device and a control program. The control program acts like a keyboard; that is, anything written to the control program appears on the pseudoterminal device as if the keystrokes had been typed in at a physical terminal. The control program also acts like a viewport to the pseudoterminal device; that is, the control program reads anything that is written by the system to the pseudoterminal device.

A pseudoterminal allows an application to be set up on the control side of the link to communicate with another application that is on the pseudoterminal side. This arrangement allows development of applications that either simulate users or monitor the communication between a real user (at a physical terminal) and an application. As with other devices, the work of the pseudoterminal is performed by a device driver and is tightly coupled to the operating system.

The VMS pseudoterminal driver software includes a set of control connection routines. Applications can use these routines to perform pseudoterminal operations and functions. Appendix C provides the VAX calling standards for these routines.

9.1 Pseudoterminal Operations

This section contains information on the following pseudoterminal operations:

- Creating a pseudoterminal
- Canceling a request
- Deleting a pseudoterminal

9.1.1 Creating a Pseudoterminal

To create a pseudoterminal, use the PTD$CREATE routine described in Appendix C. When a pseudoterminal is created, it inherits the current system terminal default attributes unless you specify an alternate set of characteristics. In either case, you cannot use PTD$CREATE to alter the following startup attributes:

- TT$M_CRFILL is cleared. To change this attribute, issue the SET MODE $QIO function.
- TT$M_LFFILL is cleared. To change this attribute, issue the SET MODE $QIO function.

- TT$M_MODEM is cleared. This attribute cannot be changed.
- TT$M_REMOTE is cleared. This attribute cannot be changed.
- TT$M_HOSTSYNC is set. To change this attribute, issue the SET MODE $QIO function.
- TT$M_TTSYNC is set. To change this attribute, issue the SET MODE $QIO function.
- TT2$M_DMA is cleared. To change this attribute, issue the SET MODE $QIO function. Changing it does not alter the behavior of TTDRIVER or the pseudoterminal.
- TT2$M_AUTOBAUD is cleared. To change this attribute, issue the SET MODE $QIO function. Changing it does not alter the behavior of TTDRIVER or the pseudoterminal.
- TT2$M_FALLBACK is cleared. To change this attribute, issue the SET MODE $QIO function.
- TT2$M_HANGUP is cleared. To change this attribute, issue the SET MODE $QIO function.
- TT2$M_DCL_MAILBX is cleared. This attribute cannot be changed.

When you create a pseudoterminal, you can specify a repeating asynchronous system trap (AST) to be delivered when the terminal connection is freed. This AST can be supplied only when the pseudoterminal is created, and it cannot be deleted. A terminal is freed when a process logs out or deassigns the last channel to the device. The AST allows the control program to determine whether or not a user of a pseudoterminal is using it. At this point, the control program can reuse or delete the pseudoterminal by deassigning the control channel.

9.1.2 Canceling a Request

To cancel a queued control connection request, the control program uses the PTD$CANCEL routine. This routine enables the pseudoterminal driver to differentiate between control requests and terminal requests that are being canceled. This routine cannot be used to flush event notification ASTs.

9.1.3 Deleting a Pseudoterminal

To delete the pseudoterminal, the control program uses the PTD$DELETE routine. When a pseudoterminal is deleted, any process that is using the pseudoterminal (except the control process) is disconnected. If you have the TT2$M_DISCONNECT bit set in the default terminal characteristics parameter (TTY_DEFCHAR2) and virtual terminals have been enabled (see Section 8.2.2.3), you get a virtual terminal upon logging in to a pseudoterminal. In this case, the process is not logged out, but the virtual terminal is disconnected from the pseudoterminal.

The PTD$DELETE request causes any pending I/O for the control program to be aborted. It deletes any queued event notification ASTs and returns the I/O buffers to the application. It also causes the pseudoterminal unit control block (UCB) to be deleted once the reference count returns to zero.

Note: If an application exits without calling PTD$DELETE, the pseudoterminal is still deleted.

9.2 Pseudoterminal Driver Features

The terminal portion of a pseudoterminal is similar to a regular VMS terminal. The pseudoterminal driver provides the following features:

- Type-ahead
- Specifiable or default line terminators
- Special operating modes, such as NOECHO and PASTHRU
- Escape sequence detection
- Terminal/mailbox interaction
- Terminal control characters, such as Ctrl/S and Ctrl/Q for starting and stopping output, Ctrl/O for discarding output, and all other special characters that are handled by the standard VMS terminal driver
- Limited full-duplex operation (simultaneously active read and write requests)

For more information on these features, see Section 8.2.

9.3 Pseudoterminal Driver Device Information

The pseudoterminal inherits its device characteristics from the system default parameters, with the following exceptions:

- The device inherits initial device characteristics from the SYSGEN-supplied default values. You can modify the device characteristics during device creation by supplying new characteristics.
- The HOSTSYNC terminal characteristic is always set.
- The device is set to NOMODEM and cannot be set to MODEM.
- The device is set not to time output character transmission. Hardware controllers time output character transmission to determine whether the controller is broken.

You can obtain information on pseudoterminal characteristics by using the Get Device/Volume Information ($GETDVI) system service, as described in Section 8.3, and the *VMS System Services Reference Manual*. For pseudoterminals other than the template unit FTA0, you can also use the sense mode functions described in Section 8.4.5 to read terminal characteristics, and the set mode function described in Section 8.4.3 to change terminal characteristics.

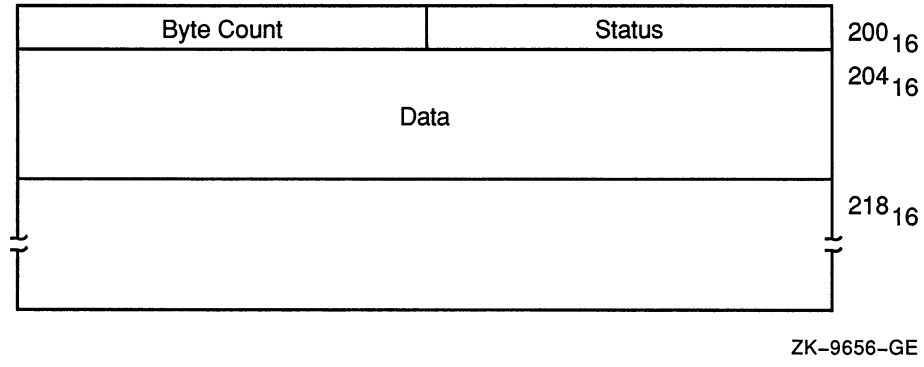
9.4 I/O Buffers

When you create a pseudoterminal, you must provide at least one page to be used as an I/O buffer. You should allocate no more than six I/O buffers for each pseudoterminal. Each page becomes one I/O buffer, and no read or write request can reference more than one I/O buffer at a time. The I/O buffers must be page aligned; therefore, you should create these pages with the $EXPREG system service or the LIB$GET_VM_PAGE routine. These pages are owned by the driver until you delete the pseudoterminal. The application is responsible for managing these pages. The application cannot use buffers that are owned by another pseudoterminal; it must decide whether to delete the buffers when they are freed by the driver or to reuse them.

The I/O buffers must be valid pages in virtual address space. Creating or deleting an I/O buffer does not alter the contents of the pages. An application is free to use these buffers in any way that it chooses. A request must fit within one I/O buffer. Attempts to span an I/O buffer result in an error. Additionally, a longword of status information must fit into the I/O buffer; this limits the largest request to 508 bytes. The low-order word of the status information longword contains the status of the request. The high-order word of the status information longword contains the actual number of bytes that are read or written.

Assume that an I/O buffer starting at 200 hexadecimal is available for use. If you want to read 20 bytes from the pseudoterminal, the **readbuf** address would be 200, and the **readbuf_len** would be 20. An application can use the rest of this buffer for other purposes, including reading or writing to the pseudoterminal. Figure 9–1 shows how the buffer would look.

Figure 9–1 Buffer Layout



9.5 Pseudoterminal Functions

This section discusses the following pseudoterminal functions:

- Reading data

- Writing data
- Using write with echo
- Flow control
- Event notification

9.5.1 Reading Data

To read data from the pseudoterminal, the control program uses the PTD$READ routine. The read request completes with a minimum of 1 character and a maximum of the number of characters requested. The read operation completes when the pseudoterminal has characters to output. If a read request is issued and no data is available, the read request is queued and then completed at a later time.

An application that issues an asynchronous pseudoterminal read can use the $SYNCH system service to find out when the read completed. The **efn** argument for the $SYNCH service must be the same as the **efn** specified in the original PTD$READ call, and the **iosb** for the $SYNCH service must match the **readbuf** of the PTD$READ call.

9.5.2 Writing Data

To write data to the pseudoterminal, the control program uses the PTD$WRITE routine. The largest write possible is 508 bytes. The write request allows you to specify a buffer to receive any output generated by the write; you do not need to issue a separate read request to read this data. Using an echo buffer allows a control application to reduce significantly the number of I/O requests required.

An application can issue only one write request at a time. Once the write request completes, the application must check the write buffer status longword to see whether all the data supplied was written. If not, the application must issue additional write requests until all the data has been accepted.

9.5.3 Using Write with Echo

If a read request is pending when a write-with-echo request is issued, the echo data is placed in the echo buffer. If more data is echoed than can fit in the echo buffer, the remaining data is placed in the pending read requests buffer. If no pending read exists, the data is held by the driver until another request that can take the data is issued. Both the read and the write with echo must use completion ASTs to allow the driver to report request completions to the application in the correct order.

If an application is not using the write-with-echo capability, the application should avoid using completion ASTs if possible. Unnecessary use of completion ASTs significantly increases the number of instructions needed to complete a read or write operation.

When using write with echo, both the **wrtbuf** and **echobuf** arguments contain I/O status information. An application must check both of these status longwords if the PTD$WRITE completes successfully. If a write operation wrote no characters, characters might still be in the echo buffer. If no data was echoed, the status in the **echobuf** is SS$_NORMAL with zero bytes transferred.

9.5.4 Flow Control

By default, the driver attempts to notify the control program of data overrun or loss. The pseudoterminal sends an XOFF AST when the type-ahead buffer is getting full. Once the pseudoterminal delivers an XOFF AST, the pseudoterminal also returns a status of SS$_DATAOVERUN with the actual number of characters input. This prevents a single request from flooding the type-ahead buffer. If a control program makes repeated attempts to insert data after receiving the SS$_DATAOVERUN message, it can flood the terminal type-ahead buffer. When the type-ahead buffer has filled, the pseudoterminal returns the status of SS$_DATALOST.

If the control program is writing to the terminal or terminal driver, it should let the terminal and terminal driver handle flow control. To do this, the application should enable all three input flow control notification ASTs. The control program should write a DC1 to the terminal if an XON AST is delivered. It should write a DC3 to a terminal if an XOFF AST is delivered, and write a bell character to the terminal if the bell AST is delivered. These signals allow the terminal to decide what to do with the flow control data. The application should ignore the SS$_DATAOVERUN and SS$_DATALOST return status and continue writing data to the pseudoterminal.

9.5.5 Event Notification

This section describes how the pseudoterminal driver provides notification of important driver events.

9.5.5.1 Input Flow Control

The driver provides three ways to indicate when the class driver wants to stop input and one way to signal when it is safe to resume output.

1. The driver returns a status of SS$_DATAOVERUN and the number of characters input for the control program write.
2. The control program can enable a BELL attention AST to be delivered when the class driver calls the PTD$SET_TERMINAL_NOTIFICATION routine. This AST is delivered if the pseudoterminal does not have the HOSTSYNC attribute set. If only a BELL or only an XOFF AST event is enabled and an XOFF or a BELL AST needs to be delivered, the AST that is available is delivered.
3. The control program can enable an XOFF attention AST to be delivered when the class driver calls the PTD$SET_TERMINAL_NOTIFICATION routine. This AST is delivered if the pseudoterminal has the HOSTSYNC attribute set.

4 The control program can enable an XON attention AST to be delivered when the class driver calls the PTD$SET_TERMINAL_NOTIFICATION routine. This AST is delivered only if the pseudoterminal has the HOSTSYNC attribute set.

9.5.5.2 Output Stop

The Output Stop AST tells the control program that the terminal driver is stopping output. This keeps the control program from having to determine whether an XOFF written to the control side is being treated by the terminal driver as flow control or data.

9.5.5.3 Output Resume

The Output Resume AST tells the control program that the terminal driver wants to resume output. This AST can be delivered at any time, even if output is active or has previously been stopped. The control program should always restart output processing when it receives this AST.

9.5.5.4 Characteristics Changed

The Characteristics Changed AST tells the control program that the terminal driver has called the pseudoterminal CHANGE CHARACTERISTICS routine. This routine is called whenever the terminal driver has changed the device characteristics. The control program should then read the pseudoterminal characteristics to determine what has changed.

9.5.5.5 Output Abort

The Output Abort AST tells the control program that the terminal driver has called the pseudoterminal ABORT OUTPUT routine. This routine is called when the terminal driver wants to flush any outstanding output data. The control program should flush any internally buffered data when this AST is received.

9.5.5.6 Terminal Driver Read Events

Three special event types notify the control program when a terminal read request starts and finishes. By default, the pseudoterminal does not deliver the read notification ASTs associated with these events. The PTD$SET_EVENT_NOTIFICATION routine must be used explicitly to enable or disable their delivery.

- Start Read—Tells the control program that the terminal driver is starting a read request. Some applications require this in order to know when to start inputting a logged session script.
- Middle Read—Tells the control program that the terminal driver has finished writing the prompt string if one was supplied.
- End Read—Tells the control program that the terminal driver has finished a read request.

Once an event notification AST is enabled, it continues to be delivered until it is canceled, or until the device is deleted. This characteristic allows the control program to enable the AST once, thus greatly reducing the risk of missing multiple rapid occurrences of an event. If the driver cannot get sufficient resources to deliver the notification AST, that report is lost.

Only one AST per event is allowed, and attempts to specify multiple ASTs result in use of the last one specified.

To enable or disable event notification, the control program uses the PTD$SET_EVENT_NOTIFICATION routine, which is described in Appendix C.

9.6 Pseudoterminal Driver Programming Example

Example 9–1 illustrates how to use the pseudoterminal. This section begins with a brief overview of the example. The example itself briefly discusses each module; the pseudocode for that module follows its discussion.

The scenario chosen for this example is a simple terminal session logging utility that uses most of the pseudoterminal capabilities. This example also illustrates how to use the write-with-echo capability, which provides a significant gain in performance.

9.6.1 Design Overview

The design approach writes the log record in a main loop that hibernates when it has no work to do. The loop uses ASTs to read keystrokes from the terminal, write to the pseudoterminal, and write data to the terminal. When a block of characters is written to the terminal, that block is placed into a queue of blocks to be written to the log file, and a wake request is issued. Logging is stopped if you log out of the subprocess, if you enter the stop logging character Ctrl\, or if a severe error occurs during data processing. When any of these events occur, all outstanding log records are written before the program exits.

One major design consideration is how flow control should be handled—either by attempting to enforce flow control, or by letting the terminal and terminal driver handle it. In this example, the terminal and terminal driver handle flow control; the driver sends XON, XOFF, or BELL characters to the terminal as necessary.

One of the six I/O buffers is permanently reserved as the terminal read buffer. This buffer is passed directly to the terminal read $QIO. This eliminates having to move data that is read from the terminal into the read buffer. The other five buffers are placed in a queue and are allocated and deallocated as needed. This pool of buffers reserves the first two longwords to be used as queue headers and traditional IOSBs. The third longword and the I/O status longwords are used by the pseudoterminal driver.

Example 9–1 Sample Pseudocode for Pseudoterminal Driver Program

```
/*
** Main Routine
**
** Function: Intitializes the environment and then hibernates, waiting
** to be awakened.  When awakened, the program checks to see whether it
** is exiting, or whether more log data is available.  If more data is
** available, the data is appended to the current log record and checked
** to see whether a log record should be written.  A log record is written
** either when maxbuf characters are in the log buffer,
** or when it finds a <CR><LF> character pair.  The algorithm
** allows an unlimited number of <NULL> fill characters to occur
** between the <CR> and the <LF>. If the program is
** exiting, it closes the log file,  deletes the pseudoterminal, resets the
** terminal, and exits.
*/
Initialize environments (This includes creating pseudoterminal, the log file
                         and starting up the subprocess.)

If (Initialization OK) Then
   Do
      while (I/O buffer to log)
         Data size = number of bytes in I/O buff
         For all data in I/O buffer
            If (cr_seen) Then
               If (current char == <LF>) Then
                  write current log buffer
                  reset cr_seen
                  point to start of log buffer
               Else if (current char != <NULL>) Then
                  insert <CR> and current char into log buffer
                  move log buffer ptr over 2 characters
                  reset cr_seen
               Endif
            Else if (current character != <CR>) Then
               insert character into log buffer
               move log buffer ptr over 1 character
            Else
               set cr_seen
            Endif

            If (log buffer ptr >= IOC$GW_MAX-48) Then
               write log buffer
               reset log buffer pointer
               reset cr_seen
            Endif
         Endloop
         Free I/O buffer call free_io_buffers
      Endwhile
      If (not exiting) Then
         Wait for more to do call SYS$HIBER
      Endif
   Until ((exiting) and (no I/O buffers to log))
```

(continued on next page)

Example 9–1 (Cont.) Sample Pseudocode for Pseudoterminal Driver Program

```
   close log file
   If ((close failed) and (exit reason is SS$_NORMAL)) Then
      set exit to status to failure reason
   Endif
   If (subprocess still running) Then
      call SYS$FORCEX to run down the subprocess
   Endif
   call PTD$CANCEL to flush all pending pseudoterminal read requests
   call SYS$CANCEL to flush all terminal requests
   call PTD$DELETE to delete the pseudoterminal
   If ((delete failed) and (exit reason is SS$_NORMAL)) Then
      set exit to status to failure reason
   Endif
   reset terminal to startup condition using SYS$QIOW
   If ((terminal reset failed) and (exit reason is SS$_NORMAL)) Then
      exit to status to failure reason
   Endif
Endif
call LIB$SIGNAL and report exit reason
Exit
/*
**
** Initialization Code
**
** Function: This routine sets the terminal characteristics, creates the
** pseudoterminal, starts up the subprocess, and opens the log file.  If
** any of these steps fail, the program undoes any steps already done and
** returns to the  main routine.
**
*/

read the maximum buffer size from IOC$GW_MAXBUF
Assign a channel to SYS$INPUT
If (assign ok) Then
   Read the terminal characteristics from the terminal
   If (read of terminal characteristics ok) Then
      Open log file with maximum record size of IOC$GW_MAXBUF
      If (open ok) Then
         Create the pseudoterminal with characteristics of terminal
         If (create ok) then
            Place 4 of the buffers on the queue of free I/O buffers
            Copy terminal characteristics and modify them to NOECHO and PASTHRU
            Set the terminal characteristics use modified value
            If (set ok) Then
               Get device name of pseudoterminal use SYS$GETDVI
               If (get ok) Then
                  Create subprocess
                  If (create ok) Then
                     Enable XON, XOFF, BELL, SET_LINE event notification ASTs
                     If (AST setup OK) Then
                        Call PTD$READ to start reading from the pseudoterminal
                                    ASTADR = ft_read_ast
                                    ASTPRM = buffer address
                                    READBUF = I/O buffer + 8
                                    READBUF_LEN = 500
                        If (read ok) Then
                           Call SYS$QIO and read a single character from the
                                       keyboard ASTADR = kbd_read_ast
```

(continued on next page)

Example 9–1 (Cont.) Sample Pseudocode for Pseudoterminal Driver Program

```
                        If (read failed) Then
                           Call PTD$CANCEL to flush queued pseudoterminal read
                           Call PTD$DELETE to delete pseudoterminal
                           Reset terminal to original state
                           Close log file and delete it
                        Endif
                     Else
                        Call PTD$DELETE to delete pseudoterminal
                        Reset terminal to original state
                        Close log file and delete it
                     Endif
                  Else
                     Call PTD$DELETE to delete pseudoterminal
                     Reset terminal to original state
                     Close log file and delete it
                  Endif
               Else
                  Call PTD$DELETE to delete pseudoterminal
                  Reset terminal to original state
                  Close log file and delete it
               Endif
            Else
               Call PTD$DELETE to delete pseudoterminal
               Close log file and delete it
            Endif
         Else
            Close log file and delete it
         Endif
      Endif
   Endif
Endif
/*
** kbd_read_ast
**
** Function: This routine is called every time data is read from the terminal.
** If the program is exiting, then the routine exits without restarting the
** read.   The character read is checked to see if the terminate processing
** character Ctrl\ was entered.  If the terminate processing character was
** entered, the exiting state is set and a SYS$WAKE is issued to start the
** main routine.  Now an attempt is made to obtain an I/O buffer in which
** to store echoed output.  If an I/O buffer is unavailable, a simple
** PTD$WRITE is issued; a PTD$WRITE with echo is issued if a buffer is
** available.  If the write completes successfully, another read is issued
** to the keyboard.
**
*/
```

(continued on next page)

Example 9–1 (Cont.) Sample Pseudocode for Pseudoterminal Driver Program

```
If (not exiting) Then
   If (read ok) Then
      Search input data for Ctrl\
      Allocate a read buffer call allocate_io_buffer
      If (got a buffer) Then
         Call PTD$WRITE to write characters to pseudoterminal
                        ASTADR = ft_echo_ast
                        ASTPRM = allocated I/O buffer
                        WRTBUF = read I/O buffer
                        WRTBUF_LEN = number of characters read
                        ECHOBUF = allocated I/O buffer
                        ECHOBUF_LEN = 500
      Else
         Call PTD$WRITE to write characters to pseudoterminal
                        WRTBUF = read I/O buffer
                        WRTBUF_LEN = number of characters read
      Endif
      If (write setup ok)
         If ((write status is ok) or (write status is SS$_DATALOST))
            Issue another single character read to terminal with
                     ASTADR = kbd_read_ast, with data going to read I/O buffer
            If (read setup failed) Then
               Set exit flag
               Set exiting reason to SS$_NORMAL
            Endif
         Else
            Set exit flag
            Set exiting reason to SS$_NORMAL
         Endif
      Else
         Set exit flag
         Set exiting reason to SS$_NORMAL
      Endif
   Else
      Set exit flag
      Set exiting reason to read failure status
   Endif
   If (exiting) Then
      Wake the mainline call SYS$WAKE
   Endif
Endif
```

(continued on next page)

Example 9–1 (Cont.) Sample Pseudocode for Pseudoterminal Driver Program

```
/*
** terminal_output_ast
**
** Function: This routine is called every time an I/O buffer is written
** to the terminal.  If the terminal write request completes successfully,
** it inserts the I/O buffer into the queue of I/O buffers to be logged.
** If the I/O buffer is the only entry on the queue, it issues a SYS$WAKE
** to start the main routine. To prevent spurious wake requests,
** SYS$WAKE is not issued if multiple entries are already on
** the queue.  If a terminal write error occurs, the routine sets the
** exit flag and wakes the main routine.
**
*/
If (terminal write completed ok) Then
   insert I/O buffer onto logging queue
   If (this is only entry on queue) Then
      wake the mainline call SYS$WAKE
   Endif
Else
   set exit flag
   set exiting reason to terminal write error reason
   wake the mainline call SYS$WAKE
Endif
/*
**
** ft_read_ast
**
** Function: This routine is called when a pseudoterminal read request
** completes.  It writes the buffer to the terminal and attempts to start
** another read from the pseudoterminal.  If the program is not exiting,
** this routine writes the buffer to the terminal and does not start another
** pseudoterminal read.
**
*/
If (not exiting)
   If (Pseudoterminal read ok) Then
      write buffer to the terminal ASTADR = terminal_output_ast
      If (write setup ok) Then
         Allocate another read buffer call allocate_io_buffer
         If (got a buffer) Then
            Call PTD$READ to restart reads from the pseudoterminal.
                     ASTADR = ft_read_ast
                     ASTPRM = buffer address
                     READBUF = I/O buffer + 8
                     READBUF_LEN = 500
            If (read setup failed) Then
               Set exit flag
               Set exiting reason to read failure reason
               Wake the mainline call SYS$WAKE
            Endif
         Else
            Set read stopped flag
         Endif
      Else
         Set exit flag
         Set exiting reason to terminal write failure reason
         Wake the mainline call SYS$WAKE
      Endif
```

(continued on next page)

Example 9–1 (Cont.) Sample Pseudocode for Pseudoterminal Driver Program

```
   Else
      Set exit flag
      Set exiting reason to terminal read failure reason
      Wake the mainline call SYS$WAKE
   Endif
Endif
/*
**
** ft_echo_ast
**
** Function: This routine is called if a write to the pseudoterminal used
** an ECHO buffer.  If any data was echoed, the output is written to the
** terminal.  If no data was echoed, the I/O buffer is freed so it can be
** used later. If the program is exiting, this routine exits.
**
*/
If (not exiting) Then
   If (ECHO buffer has data) Then
      Write the buffer to the terminal with ASTADR = terminal_output_ast
      If (error setting up write) Then
         Set exit flag
         Set exiting reason to write failure reason
         Wake mainline call SYS$WAKE
      Endif
   Else
      Free I/O buffer call free_io_buffers
   Endif
Endif
/*
** free_io_buffers
**
** Function: This routine places a free I/O buffer on the queue of available
** I/O buffers. It also restarts any stopped read operations from the
** pseudoterminals.  This routine disables AST delivery while it is running
** in order to synchronize reading and resetting the read stopped flag.
**
*/
If (not exiting) Then
   Disable AST deliver using SYS$SETAST
   If (Pseudoterminal reads not stopped) Then
      Insert I/O buffer on the interlocked queue of free I/O buffers
   Else
      Call PTD$READ to restart reads from the pseudoterminal.
                  ASTADR = ft_read_ast
                  ASTPRM = buffer address
                  READBUF = I/O buffer + 8
                  READBUF_LEN = 500
      If (no error starting read) Then
         Clear read stopped flag
      Else
         Set exit flag
         Set exit reason to read setup reason
      Endif
  Endif
  Enable AST delivery using SYS$SETAST
Endif
```

(continued on next page)

Example 9–1 (Cont.) Sample Pseudocode for Pseudoterminal Driver Program

```
/*
**
** allocate_io_buffer
**
** Function: This routine obtains a free I/O buffer from the queue of
** available I/O buffers.  If the program is exiting, this routine
** returns an SS$_FORCEDEXIT error.
**
*/
If (not exiting) Then
   remove a I/O buffer from the interlocked queue of I/O buffers
   If (queue empty) Then
      exit with reason LIB$_QUEWASEMP
else
   exit with reason SS$_FORCEDEXIT
Endif
/*
** subprocess_exit
**
** Function: This routine is called when the subprocess has completed
** and exited.  This routine checks whether the program is already exiting.
** If not, then the routine indicates that the program is exiting and wakes
** up the main program.
**
*/
If (not exiting) Then
   indicate subprocess no longer running
   set exit status to SS$_NORMAL
   indicate exiting
   call SYS$WAKE to start up main loop
Endif
/*
** xon_ast
**
** Function: This routine is called for the pseudoterminal driver to signal
** that it is ready to accept keyboard input. The routine attempts to send
** an XON character to the terminal by sending XON DC1 using SYS$QIO.
** If the attempt fails, it is not retried.
**
*/
If (not exiting) Then
   call SYS$QIO to send a <DC1> character to the terminal
Endif
/*
** bell_ast
**
** Function: This routine is called when the pseudoterminal driver wants
** to warn the user to stop sending keyboard data.  The routine attempts
** to ring the terminal bell by sending the BELL character to the terminal
** using SYS$QIO.  If the attempt fails, it is not retried.
**
*/
If (not exiting) Then
   call SYS$QIO to send a <BELL> character to the terminal
Endif
```

(continued on next page)

Example 9–1 (Cont.) Sample Pseudocode for Pseudoterminal Driver Program

```
/*
** xoff_ast
**
** Function: This routine is called when the pseudoterminal driver wants to
** signal that it will stop accepting  keyboard input. The routine attempts
** to send an XOFF character to the terminal by sending XOFF DC3 to the
** terminal using SYS$QIO.  If the attempt fails, it is not retried.
**
*/
If (not exiting) Then
   call SYS$QIO to send a <DC3> character to the terminal
Endif
/*
** set_line_ast
**
** Function: This routine is called when the pseudoterminal device
** characteristics  change.  The routine reads the current pseudoterminal
** characteristics, changes the characteristics to set PASTHRU and NOECHO,
** and applies the characteristics to the input terminal.  If the attempt
** to alter the terminal characteristics fails, it is not retried.
**
*/
If (not exiting) Then
   call SYS$QIOW to read the pseudoterminals characteristics
   If (not error) Then
      Set the alter the just read characteristics to have PASTHRU and NOECHO
      attributes
      call SYS$QIO to set the terminal characteristics.
   Endif
Endif
```

10 Shadow-Set Virtual Unit Driver

This chapter describes the use of the shadow-set virtual unit driver (SHDRIVER) and provides an overview of VMS Phase II Volume Shadowing. For more detailed information on VMS Phase II Volume Shadowing, see the *VMS Volume Shadowing Manual.*

10.1 Introduction

VMS Phase II Volume Shadowing allows shadowing on the same configurations as phase I volume shadowing described in the *VAX Volume Shadowing Manual.* In addition, phase II supports clusterwide shadowing of all MSCP (mass storage control protocol) compliant DSA (DIGITAL Storage Architecture) disks that have the same physical geometry, on a single system or located anywhere in a VAXcluster system.

Phase II volume shadowing supports clusterwide shadowing of all DSA devices. Phase II is not limited to HSC (hierarchical storage controller) disks, but extends volume shadowing capabilities to all DSA disks including those on local adapters, all DSSI (DIGITAL Small Systems Interconnect) RF-series disk devices on any VAX computer, all interfaces (computer interconnect [CI], Ethernet, mixed-interconnect), and across VMS MSCP servers.

SHDRIVER is the driver that controls the virtual unit functions described in Section 10.4.

Like phase I shadowing, any given phase II shadow set can have a maximum of three shadow set members. Phase II shadowing also provides more flexibility regarding shadow set membership because phase II shadowing does not limit the number of shadow sets or shadow set members for each controller or pair of controllers.

Note: Phase II volume shadowing places no restrictions on the total number of shadow set members; however, interconnect or adapter bandwidths during shadow-set copy operations might force shadow set limits.

Other phase II volume shadowing features include:

- Controller independence. Shadow set members can be located on any node in a VAXcluster system that has volume shadowing enabled.
- Clusterwide, homogeneous shadow-set maintenance functions.
- Ability to survive controller, disk, and media failures transparently.
- Shadowing functions do not affect application I/O.
- Coexistence with phase I volume shadowing. Shadow sets are differentiated by virtual-unit name format.

10.2 Phase I and Phase II Compatibility

You can use both phase I and phase II shadowing on a standalone system or a VAXcluster system. For example, you can use phase I shadowing on one node and phase II shadowing on another node. Note, however, that members in a given shadow set must be either phase I or phase II; they cannot be of mixed types. Also, using two types of shadowing on the same system complicates system management because:

- The method of naming virtual units differs between the phase I and phase II products.
- The SYSGEN parameter settings for a given node must be set to the shadowing mode you choose to use for that node.

Although the underlying design of phase II volume shadowing is different from that of phase I, the user interface for the two shadowing products is very similar. The VMS Mount Utility commands, SHOW commands, and system services for both shadowing products are very similar, and application I/O semantics for shadow set manipulation is the same for both. Phase II shadowing supports a new MOUNT qualifier that is useful for automating the rebuilding of former shadow sets.

For discussions of phase II SYSGEN parameters, booting, system upgrades, and migration and compatibility considerations, see the *VMS Volume Shadowing Manual.*

10.3 Configurations

VMS Phase II Volume Shadowing does not depend on specific hardware in order to operate. All shadowing functions can be performed on VAX computers running the VMS operating system. Shadowing capabilities are supported on all VMS MSCP devices, including RF devices, and on all types of DSA-compliant disks. Shadow set members must have the same physical geometry (that is, the same number of identical logical blocks [LBNs]), and members can be located anywhere in a VAXcluster system.

10.3.1 Processors and Controllers

Volume shadowing requires a minimum of one VAX computer, one MSCP-compliant mass storage controller, and at least two DSA disk drives and volumes.

Digital provides shadowing capabilities on:

- All DSA and MSCP-compliant drives:
  - — On the same local controller
  - — On different local controllers
  - — On controllers local to different VMS hosts and to VMS MSCP served to the VAXcluster system over CI, Ethernet, DSSI, and mixed-interconnect configurations

- DSA- and MSCP-compliant integrated storage elements (ISEs) on the same DSSI as the VMS hosts
- DSA- and MSCP-compliant ISEs that are VMS MSCP served to the VAXcluster system on the same or different DSSIs
- All DSA disks having the same physical geometry
- All interfaces for DSA devices

Devices that cannot be shadowed include:

- Small Computer System Interface (SCSI) devices
- MicroVAX 2000 RD disks
- Pre-DSA disk devices (such as MASSBUS, RK07, RL02, RPO6, and RP07)

Table 10–1 lists the hardware supported by VMS Volume Shadowing.

Table 10–1 Hardware Devices That Support Volume Shadowing

Disk Controllers and Adapters
HSC40, HSC50, HSC70
KDA50
KDB50
KDM70
RQDX2, RQDX3[1]
UDA50
Disks
ESE20
RA60, RA70, RA80, RA81, RA82, RA90, RA92
RD51, RD52, RD53, RD54 (when connected to RQDX)
RF30, RF31, RF71, RF72

[1] Shadow set members should be on different RQDX controllers.

10.3.2 Compatible Disk Drives and Volumes

Volume shadowing requires compatibility among the physical units (disk drives and volumes) that form a shadow set. For instance:

- Units must have the same geometry, including the same number of sectors per track, the same number of tracks per cylinder, and the same number of cylinders per volume.
- Units and controllers must conform to DSA and MSCP.

- Units should not have hardware write protection enabled. Hardware write protection stops the volume shadowing software from maintaining identical volumes.

10.4 Driver Functions

This section describes the major virtual unit functions supported by SHDRIVER. In addition to the virtual unit functions described in this section, SHDRIVER supports all VMS disk functions. SHDRIVER receives QIO operations from application programs and is a client of the disk class drivers DUDRIVER and DSDRIVER. Applications access the shadow set as they would access a standard VMS disk.

Table 10–2 summarizes the major SHDRIVER functions. The subsections that follow describe these functions in detail.

Table 10–2 Functions of the Shadow Set Virtual Unit Driver

Function	Description
CRESHAD	Creates a virtual unit
ADDSHAD	Evaluates a physical member and adds members
COPYSHAD	Triggers and controls copy operations
REMSHAD	Removes a physical member
AVAILABLE	Virtual unit dissolution
SENSECHAR	Verifies shadow set status
READ	Directs I/O to a physical member
WRITE	Propagates a write operation to all physical members

10.4.1 CRESHAD

The CRESHAD function creates a virtual unit, and establishes a clusterwide lock. This function is internal and can be issued only by the MOUNT utility, from either DCL or the $MOUNT system service.

The function code is:

IO$_CRESHAD

IO$_CRESHAD takes the function modifier IO$M_EXISTS. When IO$M_EXISTS is specified, the virtual unit exists in the cluster and IO$_CRESHAD creates an identical, multimember set. If IO$M_EXISTS is not set, the shadow set does not exist yet and IO$_CRESHAD creates a single member shadow set.

IO$_CRESHAD also validates the shadow-set virtual unit, enables distributed locking protocols, and creates or updates the unit control block (UCB) and shadow set (SHAD) structures for the virtual unit.

The following are the device-dependent or function-dependent arguments for IO$_CRESHAD. (Note that these arguments are internal; they cannot be accessed by user programs.)

- P1—The address of the shadow-set virtual unit name string, or zero.
- P2—If P1 does not equal zero, this parameter is the size of the shadow set virtual unit name string. If P1 equals zero, this parameter is the unit number of the shadow-set virtual unit.
- P3—The storage control block (SCB) logical block number (LBN) from the first member listed in the MOUNT command.

IO$_CRESHAD can return the following status codes:

- SS$_NORMAL—Normal successful completion.
- SS$_ACCVIO—An access violation occurred because the shadow-set virtual unit name string is not readable.
- SS$_INSFMEM—Not enough memory exists to allocate the UCB or SHAD structure.
- SS$_IVDEVNAM—The unit number for the shadow set virtual unit is greater than 9999.
- SS$_INCSHAMEM—The specified shadow set member cannot be a member of the existing shadow set because it is the quorum disk or is of incompatible geometry.
- SS$_DUPINIT—The specified shadow set already exists.

10.4.2 ADDSHAD

The ADDSHAD function is an internal control function that validates the channel number and unit control block (UCB) address for a proposed shadow-set virtual unit member. This function also validates the specified copy-type information for the shadow-set virtual unit member and performs a clusterwide update. IO$_ADDSHAD then adds the member to the shadow set by updating the disk data structure and notifying other shadow set members.

The function code is:

IO$_ADDSHAD

IO$_ADDSHAD takes no function modifiers.

The following are the device-dependent or function-dependent arguments for IO$_ADDSHAD:

- P2—The channel number assigned to the proposed shadow set member
- P3—The copy type (full or merge) of the member

IO$_ADDSHAD can return the following status codes:

- SS$_NORMAL—Normal successful completion.
- SS$_BADPARAM—The UCB is not a virtual unit UCB.

- SS$_ILLIOFUNC—The unit is not a shadow set virtual unit, or P2 points to a shadow set virtual unit.
- SS$_NOPRIV—The user has no access to the channel specified in P2.
- SS$_IVCHAN—The channel number specified in P2 is invalid.
- SS$_IVMODE—The copy-type mode specified in P3 is invalid.
- SS$_INSFMEM—Insufficient memory is available to allocate the shadow set member VCB, and resource wait mode is disabled.
- SS$_INCSHAMEM—The physical unit cannot be added to the shadow set for one of the following reasons:
  - The shadow set is already fully populated.
  - The device being added is not a VMS MSCP device.
  - The geometry or hardware properties of the physical unit do not match those of the other shadow set members.
  - The proposed shadow set member is the quorum disk.
  - The physical unit is already a member of another shadow set.

10.4.3 COPYSHAD

The COPYSHAD function triggers and controls copy operations. It should be issued only by the MOUNT utility or the shadow server process.

The function code is:

IO$_COPYSHAD

IO$_COPYSHAD takes the following function modifiers:

- IO$M_COPYOP—Requests a copy operation. This modifier is issued by the shadow server when the server requests a copy operation.
- IO$M_STEPOVER—Provides for protection of the volume storage control block (SCB) during copy operations.

The following are the device-dependent or function-dependent arguments for IO$_COPYSHAD:

- P2—The request byte count
- P3—The starting LBN
- P4—The copy mode

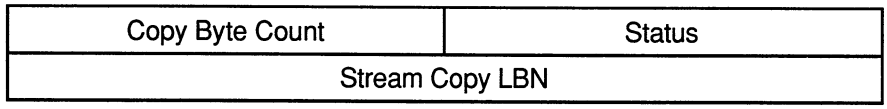
IO$_COPYSHAD can return an SS$_ILLIOFUNC status for one of the following reasons:

- A COPY I/O command was issued by a process other than the shadow server.
- Another copy operation was attempted while a copy operation was already in progress.

- A COPY I/O command was issued while a copy operation was not in progress.

Figure 10–1 illustrates the I/O status block returned for copy operations.

Figure 10–1 I/O Status Block for Copy Operations



ZK–9699–GE

IO$_COPYSHAD receives copy initiation I/O request packets (IRPs) during a copy operation. IO$_COPYSHAD then performs the full or merge copy operation; upon completion, IO$_COPYSHAD receives IO$_COPYSHAD IRPs from the shadow server. A SS$_RESET message is returned if a copy-state reconciliation is needed because the copy initialization process has triggered a state change. The shadow server then issues an IO$_SENSECHAR function (see Section 10.4.6).

10.4.4 REMSHAD

The REMSHAD function removes a device from a shadow set virtual unit. The REMSHAD function can be issued only by the $DISMOUNT system service to a physical device.

The function code is:

IO$_REMSHAD

The IO$_REMSHAD function takes no modifiers.

IO$_REMSHAD verifies that the unit to be removed is a valid physical volume and a member of a shadow set. If either of these conditions is not true, SH$REMSHAD_FDT returns an SS$_ILLIOFUNC status. IO$_REMSHAD then removes the specified device from a shadow set by breaking the links between the device UCB and the other shadow-set data structures; clears device status fields; and removes references to the device from the device array in the SHAD structure. IO$_REMSHAD also updates the SCBs of disks that remain in the shadow set after changes are made to the local data structures.

If the shadow set has too few members to remove one, a SS$_BADPARAM status code is returned. Otherwise, SH$START_REMSHAD returns SS$_NORMAL to indicate a normal successful completion. Upon successful completion, IO$_REMSHAD notifies the remaining shadow set members.

10.4.5 AVAILABLE

The AVAILABLE function makes a virtual unit available by eliminating local control of the unit. AVAILABLE also makes the necessary changes to the local shadowing I/O database for disassembling the shadow set on the node, and releases cluster shadowing locks.

The function code is:

IO$_AVAILABLE

The IO$_AVAILABLE function takes no modifiers.

IO$_AVAILABLE first unloads the FDT (function decision table) for shadowing. If the specified unit is a member of a shadow set, or if IO$V_DISSOLVE is present for a physical unit, IO$_AVAILABLE returns an SS$_ILLIOFUNC status code. IO$_AVAILABLE then makes all changes to the local shadowing I/O database that are necessary to dismount the shadow set on the node.

10.4.6 SENSECHAR

The SENSECHAR function verifies the existence and membership status of a shadow set.

The function code is:

IO$_SENSECHAR

This function code takes the following function modifiers:

- IO$M_SHADOW
- IO$M_COPYOP

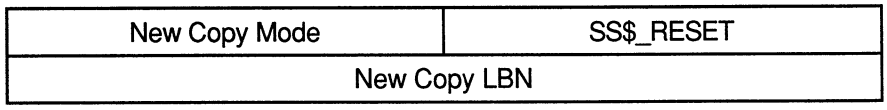
When the IO$M_SHADOW modifier is specified, the IO$_SENSECHAR verifies the presence of a properly prepared VCB, initializes the VCB if necessary, and passes the request to the driver start I/O routine. If the IO$M_SHADOW modifier is not specified, the routine returns to the FDT processing loop. An SS$_BADPARAM status code is returned if the shadow-set virtual unit VCB is not present or is set up incorrectly.

IO$_SENSECHAR processing then begins. If specified, the modifiers IO$M_SHADOW and IO$M_COPYOP are used by the shadow server to determine whether or not to do any more copy operations before deallocating its resources.

IO$_SENSECHAR then ensures that the VCB address information in the SHAD data structure is up to date.

Finally, IO$_SENSECHAR determines whether any copy requests were previously suspended. A status code of SS$_NORMAL indicates that no copy operations were suspended. A status code of SS$_RESET indicates that copy operations are to continue; the LBN and mode data are supplied in the I/O status block, as shown in Figure 10–2.

Figure 10–2 I/O Status Block for Copy Information



ZK–9698–GE

10.4.7 Read and Write Functions

With just minor changes, the read and write functions for SHDRIVER operate the same as for the disk class driver (see Sections 3.4.1 and 3.4.2).

During an SHDRIVER read operation, the VAX host directs the read to the member volume with the shortest path.

During a write operation, SHDRIVER directs the write to each member volume. The write operations for each member volume usually proceed in parallel; the virtual unit write operation terminates when all writes have completed. The write function for SHDRIVER takes the IO$M_FC_VUEX function modifier; this modifier should not be used by application programs.

The read and write SHDRIVER functions, as well as all user functions, are issued by user programs. All other SHDRIVER functions are invoked by MOUNT and DISMOUNT commands, or the $MOUNT and $DISMOUNT system services.

10.5 Error Processing

Shadow set recovery and repair are handled by volume processing, which replaces mount verification for shadow sets. The main difference in phase II shadowing is that membership failure decisions are made by the VAX hosts. Device errors that result in inaccessibility of physical member units first utilize the class driver's connection walking algorithm. If that fails, a local decision is made on the shadow set membership. The rules are:

- If some, but not all, members of the set are accessible, then the local node sequentially adjusts the membership and notifies the other hosts.
- If no members are accessible, no modifications to the set membership are made.

There are two types of volume processing: active and passive. Active volume processing handles error processing on a local node. Triggered by a failed I/O operation, active volume processing also controls mount verification functions, member removal, and failover. Passive volume processing is triggered by lock messages or by a cluster event. Passive volume processing revalidates shadow set membership, ensures that the shadow set reflects changes made outside the shadow set, and handles the following functions:

- Member additions from other nodes

- Member removals from other nodes
- A new node mounting the shadow set
- A node dismounting the shadow set
- A system crash on a node that has the shadow set mounted

For more information, see the *VMS Volume Shadowing Manual*.

11 Using the VMS Generic SCSI Class Driver

This chapter describes the use of the VMS Generic Small Computer Systems Interface (SCSI) class driver.

11.1 Overview of SCSI

The American National Standard for information systems—Small Computer System Interface–2 (SCSI–2) specification defines mechanical, electrical and functional requirements for connecting small computers to a wide variety of intelligent devices, such as rigid disks, flexible disks, magnetic tape devices, printers, optical disks, and scanners. It specifies standard electrical bus signals, timing, and protocol, as well as a standard packet interface for sending commands to devices on the SCSI bus.

Certain VAXstation and MicroVAX systems employ the SCSI bus as an I/O bus. For these systems, Digital offers SCSI-compliant disk and tape drives, such as the RZ55 300MB read/write disk, the RRD40 600MB compact disk, and the TZK50 95MB streaming tape drive. The VMS operating system also allows non-Digital-supplied devices including disk drives, tape drives, and scanners to be connected to the SCSI bus of such a system.

SCSI has been widely adopted by manufacturers for a variety of peripheral devices. However, because the ANSI SCSI standard is broad in scope, not all devices that implement its specifications can fully interrelate on the bus. Digital fully supports SCSI-compliant equipment sold or supplied by Digital. Proper operation of products not sold or supplied by Digital cannot be assured.

For more information, refer to the following documents:

- American National Standard for Information Systems—Small Computer System Interface—2 (SCSI–2) specification (X3T9.2/86–109)

  The SCSI–2 specification is a draft of a proposed standard. Until it is finally approved, copies of this document may be purchased from: Global Engineering Documents, 2805 McGaw, Irvine, California 92714, United States; or (800) 854-7179 or (714) 261-1455. Please refer to document X3.131–198X.

- American National Standard for Information Systems—Small Computer System Interface specification (X3.131–1986)

  Copies of this document may be obtained from: American National Standards Institute, Inc., 1430 Broadway, New York, New York, 10018. This document is now known as the SCSI–1 standard.

Digital publishes two additional documents to help third-party vendors prepare SCSI peripherals and peripheral software for use with Digital's workstations and MicroVAX systems.

- The *Small Computer System Interface: An Overview* (EK–SCSISOV–001) provides a general description of Digital's SCSI third-party support program.
- The *Small Computer System Interface: A Developer's Guide* (EK–SCSIS–SP–001) presents the details of Digital's implementation of SCSI within its operating systems.

11.2 VMS SCSI Class/Port Architecture

The VMS operating system employs a class/port driver architecture to communicate with devices on the SCSI bus. The class/port design allows the responsibilities for communication between the operating system and the device to be cleanly divided between two separate driver modules (see Figure 11–1).

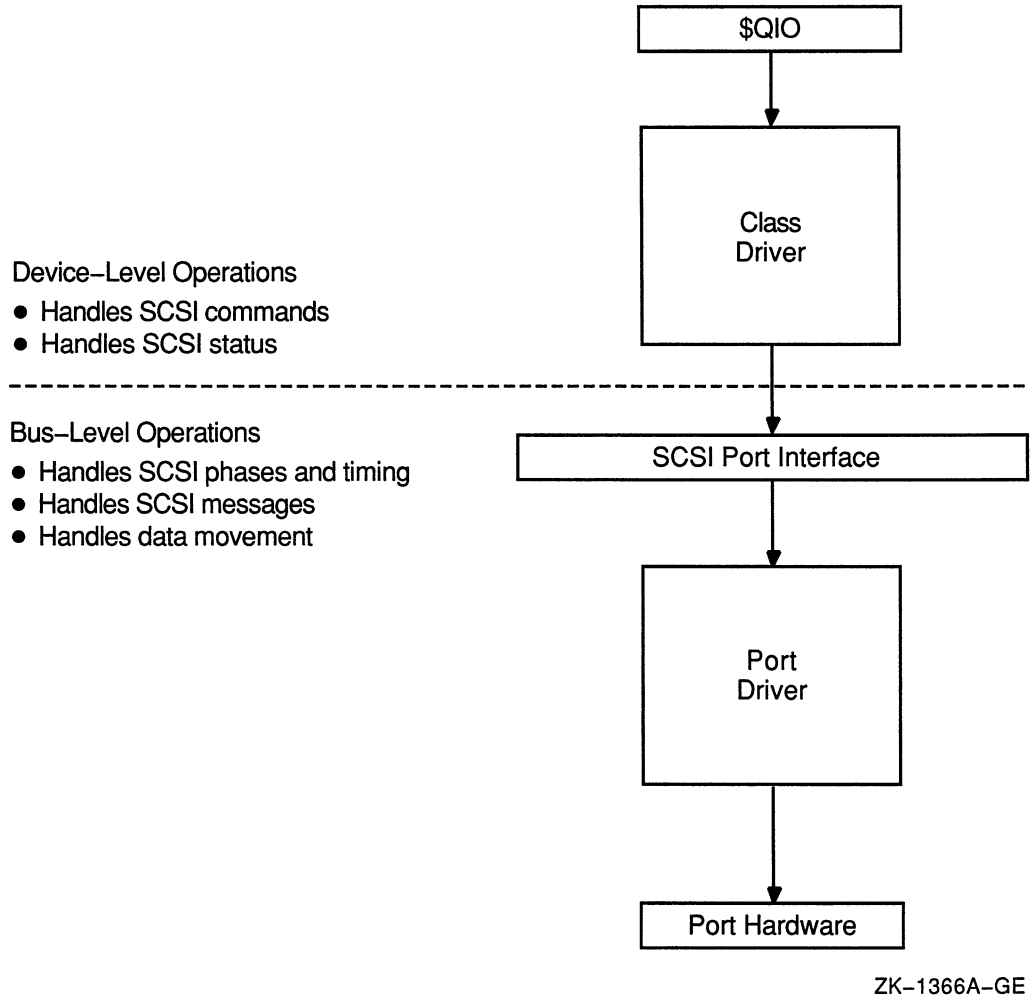
The *SCSI port driver* transmits and receives SCSI commands and data. It knows the details of transmitting data from the local processor's SCSI port hardware across the SCSI bus. Although it understands SCSI bus phases, protocol, and timing, it has no knowledge of which SCSI commands the device supports, what status messages it returns, or the format of the packets in which this information is delivered. Strictly speaking, the port driver is a communications path. When directed by a SCSI class driver, the port driver forwards commands and data from the class driver onto the SCSI bus to the device. On any given MicroVAX/VAXstation system, a single SCSI port driver handles bus-level communications for all SCSI class drivers that may exist on the system.

The *SCSI class driver* acts as an interface between the user and the SCSI port, translating an I/O function as specified in a user's $QIO request to a SCSI command targeted to a device on the SCSI bus. Although the class driver knows about SCSI command descriptor buffers, status codes, and data, it has no knowledge of underlying bus protocols or hardware, command transmission, bus phases, timing, or messages. A single class driver can run on any given MicroVAX/VAXstation system, in conjunction with the SCSI port driver that supports that system. The VMS operating system supplies a standard SCSI disk class driver and a standard SCSI tape class driver to support its disk and tape SCSI devices.

11.3 Overview of the VMS Generic SCSI Class Driver

The VMS generic SCSI class driver provides a mechanism by which an application program can control a non-Digital-supplied SCSI device that cannot be controlled by the standard VMS disk and tape class drivers. By means of a Queue I/O Request ($QIO) system service call, a program can pass to the generic SCSI class driver a preformatted SCSI command descriptor block. The generic SCSI class driver, in conjunction with the

Figure 11–1 VMS SCSI Class/Port Interface



standard VMS SCSI port driver, delivers this SCSI command to the device, manages any transfer of data from the device to a user buffer, and returns SCSI status to the application.

In effect, an application using the generic SCSI class driver implements details of device control usually managed within device driver code. The programmer of such an application must understand which SCSI commands the device supports and which SCSI status values the device returns. The programmer must also be aware of the device's timeout requirements, data transfer capabilities, and command retry behavior.

The application program sets up the characteristics of the connection the generic SCSI class driver uses when delivering commands to, exchanging data with, and receiving status from the device. The program associates each I/O operation the device can perform with a specific SCSI command. When it receives a request for a particular operation, the application

program creates the specific command descriptor block that, when passed to the device, causes it to perform that operation.

The application initiates all transactions to the SCSI device by means of a $QIO call to the generic SCSI class driver, supplying the address and length of the SCSI command descriptor block, plus the parameters of any data transfer operation, in the call. When the transaction completes and the application program regains control, it interprets the returned status value, processes any returned data, and services any failure. To avoid conflicts with other applications accessing the same device, an application may need to explicitly allocate the device.

Because the generic SCSI class driver has no knowledge of specific device errors, it neither logs device errors nor implements error recovery. An application using the driver must manage device-specific errors itself. To service an error returned on a single transaction, the application must issue additional $QIO requests and initiate further transactions to the device. If more precise or more efficient error recovery is required for a device, the developer should consider writing a third-party SCSI class driver, as described in *VMS Device Support Manual*. A third-party SCSI class driver can log errors associated with device activity by using the method described in the *VMS Device Support Manual*.

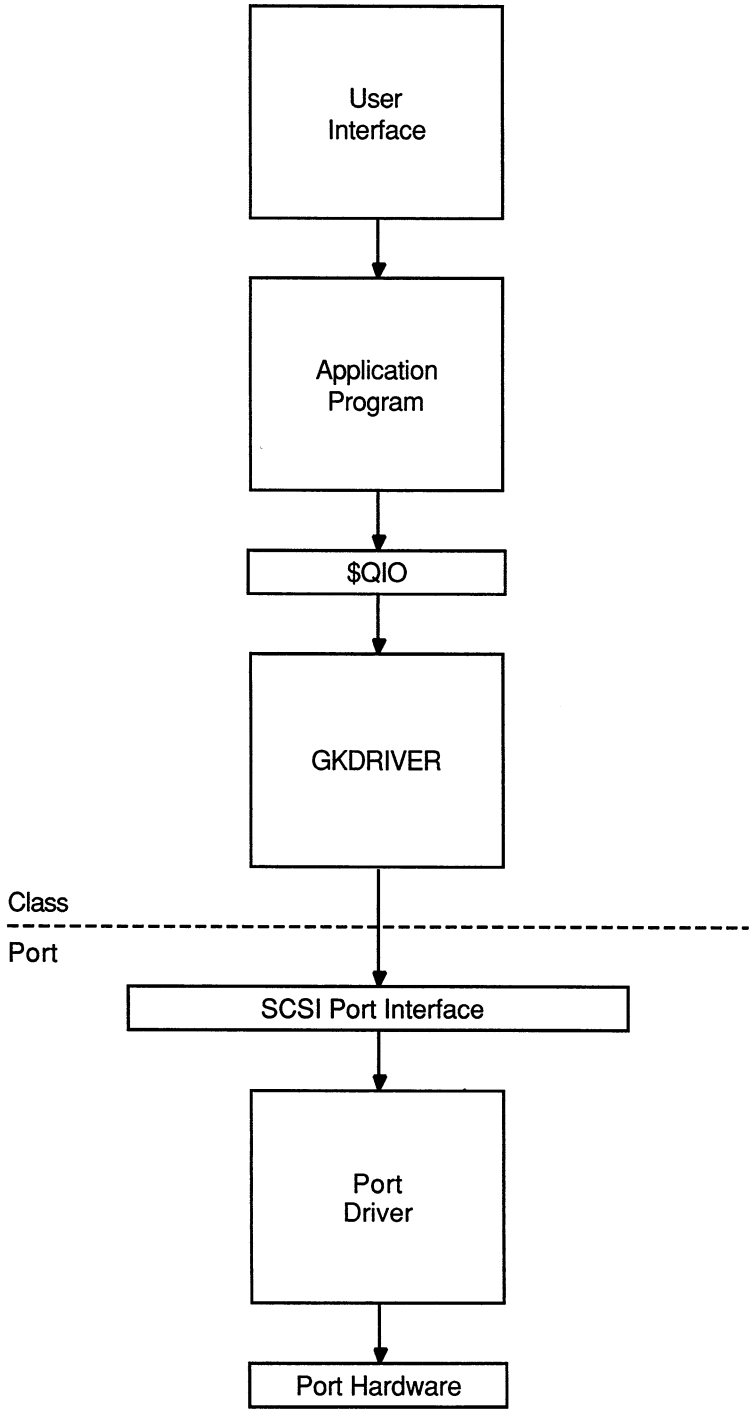
A third-party class driver is the only means of supporting devices that themselves generate transactions on the SCSI bus, such as notification of a device selection event to the host processor. See the description of asynchronous event notification (AEN) in the *VMS Device Support Manual*.

Figure 11–2 illustrates the flow of a $QIO request through the generic SCSI class driver and the port driver.

When direct access to a target device on the SCSI bus is required, the generic SCSI class driver is loaded for that device, as described in Section 11.6. An application program using the generic class driver performs the following tasks to issue a command to the target device:

1. Calls the Assign I/O Channel ($ASSIGN) system service to assign a channel to the generic SCSI class driver, and allocate the device for the application's exclusive use
2. Formats a SCSI command descriptor block
3. Formats any data to be transferred to the device
4. Calls the Queue I/O Request ($QIO) system service to request the generic SCSI class driver to send the SCSI command descriptor block to the port driver
5. Upon completion of the I/O request, interprets the SCSI status byte and any data returned from the target device

Figure 11–2 Generic SCSI Class Driver Flow



These operations are described in subsequent sections.

Note: Because incorrect or malicious use of the generic SCSI class driver can result in SCSI bus hangs and lead to SCSI bus resets, DIAGNOSE and PHY_IO or LOG_IO privileges are required to access the driver. An application program can be designed in such a way as to filter user I/O requests, thus allowing nonprivileged users access to some device functions.

11.4 Accessing the VMS Generic SCSI Class Driver

Interactive commands and procedure calls can use the VMS generic SCSI class driver to access devices on the SCSI bus. However, it is unlikely that a user application would access a device on the SCSI bus by directly using the $QIO interface of the generic SCSI class driver. First of all, any user process directly using the $QIO interface would require DIAGNOSE and PHY_IO or LOG_IO privileges. Under normal circumstances, it would be a system security risk to grant DIAGNOSE and PHY_IO or LOG_IO privileges to many system users. Secondly, it would be cumbersome for end users of the device to identify, format, and issue SCSI commands to the device. Rather, it would be more efficient to develop an interface that hides these details.

A utility program, installed with the DIAGNOSE and PHY_IO or LOG_IO privileges, can provide nonprivileged users with a command line interface to a SCSI device. The utility translates interactive commands provided by the user into the appropriate set of SCSI commands and sends them to the device using the $QIO interface provided by the generic SCSI class driver. The utility checks user commands to ensure that only valid SCSI commands are sent to the device. See the *Guide to VMS Programming Resources* and the *VMS Install Utility Manual* for information about installing images with privileges.

A privileged shareable image can provide system applications with a procedure interface to a SCSI device. The image contains a set of procedures that translate operations specified by the caller into the appropriate set of SCSI commands. The SCSI commands are sent to the device through the $QIO interface of the generic SCSI class driver. The privileged shareable image checks its caller's parameters to ensure that only valid SCSI commands are sent to the device. See the *Introduction to VMS System Services* for information about creating shareable images.

11.5 SCSI Port Features Under Application Control

The standard VMS SCSI port driver provides mechanisms by which the generic SCSI class driver can control the nature of data transfers and command transmission across the SCSI bus. An application uses the $QIO interface to tailor these mechanisms to the specific device it supports. Among the features under application program control are the following:

- Data transfer mode
- Disconnection and reselection
- Command retry

- Command timeouts

The following sections discuss these features.

11.5.1 Setting the Data Transfer Mode

The SCSI bus defines two data transfer modes, asynchronous and synchronous. In asynchronous mode, for each REQ from a target there is an ACK from the host prior to the next REQ from the target. Synchronous mode allows higher data transfer rates by allowing a pipelined data transfer mechanism where, for short bursts (defined by the REQ-ACK offset), the target can pipeline data to an initiator without waiting for the initiator to respond.

Whether or not a port or a target device allows synchronous data transfers, it is harmless for the program to set up the connection to use such transfers. If synchronous mode is not supported, the port driver automatically uses asynchronous mode.

To use synchronous mode in a transfer, a programmer using the generic SCSI class driver must ensure that both the SCSI port and the SCSI device involved in the transfer support synchronous mode. The SCSI port of the VAXstation 3520/3540 system allows both synchronous and asynchronous transfers, whereas that of MicroVAX/VAXstation 3100 systems supports only asynchronous transfers.

To set up a connection to use synchronous data transfer mode, a program using the generic SCSI class driver sets the **syn** bit in the **flags** field of the generic SCSI descriptor, the address of which is passed to the driver in the **p1** argument to the $QIO request.

11.5.2 Enabling Disconnection and Reselection

The ANSI SCSI specification defines a disconnection facility that allows a target device to yield ownership of the SCSI bus while seeking or performing other time-consuming operations. When a target disconnects from the SCSI bus, it sends a sequence of messages to the initiator that cause it to save the state of the I/O transfer in progress. Once this is done, the target releases the SCSI bus. When the target is ready to complete the operation, it reselects the initiator and sends to it another sequence of messages. This sequence uniquely identifies the target and allows the initiator to restore the context of the suspended I/O operation.

Whether disconnection should be enabled or disabled on a given connection depends on the nature and capabilities of the device involved in the transfer, as well as on the configuration of the system. In configurations where there is a slow device present on the SCSI bus, enabling disconnection on connections that transfer data to the device can increase bus throughput. By contrast, systems where most of the I/O activity is directed towards a single device for long intervals can benefit from disabling disconnection. By disabling disconnection when there is no contention on the SCSI bus, port drivers can increase throughput and decrease the processor overhead for each I/O request.

By default, the VMS class/port interface disables the disconnect facility on a connection. To enable disconnection, an application program using the generic SCSI class driver sets the **dis** bit of the **flags** field of the generic SCSI descriptor, the address of which is passed to the driver in the **p1** argument to the $QIO call.

11.5.3 Disabling Command Retry

The SCSI port driver implements a command retry mechanism, which is enabled on a given connection by default.

When the command retry mechanism is enabled, the port driver retries up to three times any I/O operation that fails during the COMMAND, Message, Data, or STATUS phases. For instance, if the port driver detects a parity error during the Data phase, it aborts the I/O operation, logs an error, and retries the I/O operation. It repeats this sequence twice more, if necessary. If the I/O operation completes successfully during a retry attempt, the port driver returns success status to the class driver. However, if all retry attempts fail, the port driver returns failure status to the class driver.

An application may need to disable the command retry mechanism under certain circumstances. For example, repeated execution of a command on a sequential device may produce different results than are intended by a single command request. A tape drive could perform a partial write and then repeat the write without resetting the tape position.

An application program using the generic SCSI class driver can disable the command retry mechanism by setting the **dpr** bit of the **flags** field of the generic SCSI descriptor, the address of which is passed to the driver in the **p1** argument to the $QIO request.

11.5.4 Setting Command Timeouts

The SCSI port driver implements several timeout mechanisms, some governed by the ANSI SCSI specification and others required by the VMS operating system. The timeouts required by the VMS operating system include the following:

Timeout	Description
Phase change timeout	Maximum number of seconds for a target to change the SCSI bus phase or complete a data transfer. (This value is also known as the *DMA timeout*.) Upon sending the last command byte, the port driver waits this many seconds for the target to change the bus phase lines and assert REQ (indicating a new phase). Or, if the target enters the DATA IN or DATA OUT phase, the transfer must be completed within this interval.

Timeout	Description
Disconnect timeout	Maximum number of seconds, from the time the initiator receives the DISCONNECT message, for a target to reselect the initiator so that it can proceed with the disconnected I/O transfer.

An application program using the generic SCSI class driver is responsible for maintaining both of these timeout values. It has the following options:

- Accepting a connection's default value. The default value for both timeouts is 20 seconds.
- Altering the connection's default value. To modify the default values, the class driver specifies nonzero values for the **phase change timeout** and **disconnect timeout** fields of the generic SCSI descriptor, the address of which is passed to the driver in the **p1** argument to the $QIO system service call.

11.6 Configuring a Device Using the Generic Class Driver

The System Generation Utility (SYSGEN) loads the generic SCSI class driver into system virtual memory, creates additional data structures for the device unit, and calls the driver's controller initialization routine and unit initialization routine. SYSGEN automatically loads and autoconfigures the SCSI port driver at system initialization. As part of autoconfiguration, SYSGEN polls each device on each SCSI bus. If the device identifies itself as a direct-access device, direct-access CDROM device, or flexible disk device, SYSGEN automatically loads the VMS disk class driver (DKDRIVER); if the device identifies itself as a sequential-access device, SYSGEN automatically loads the VMS tape class driver (MKDRIVER). If the autoconfiguration facility does not recognize the type of the SCSI device, it loads no driver.

Consequently, if a non-Digital-supplied SCSI device requires that the generic class driver be loaded, it must be configured by an explicit SYSGEN CONNECT command, as follows:

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> CONNECT GKpd0u /NOADAPTER
```

In this command, *GK* is the device mnemonic for the generic SCSI class driver (GKDRIVER); *p* represents the SCSI port ID (for instance, the controller ID *A* or *B*); *d* represents the SCSI device ID (a digit from 0 to 7); 0 signifies the digit zero; and *u* represents the SCSI logical unit number (a digit from 0 to 7).

Multiple devices residing on any SCSI bus in the system can share GKDRIVER as a class driver, as long as a SYSGEN CONNECT command is issued for each target device that requires the driver.

Because just one connection can exist through the SCSI port driver to each target, the generic class driver cannot be used for a target if a different SCSI class driver is already connected to that target. For example, if the SCSI disk class driver has a connection to device ID 2 on the SCSI bus identified by SCSI port ID *B* (DKB200), the generic class driver cannot be

used to communicate with this disk. An attempt to connect GKDRIVER for this target results in GKB200 being placed off line.

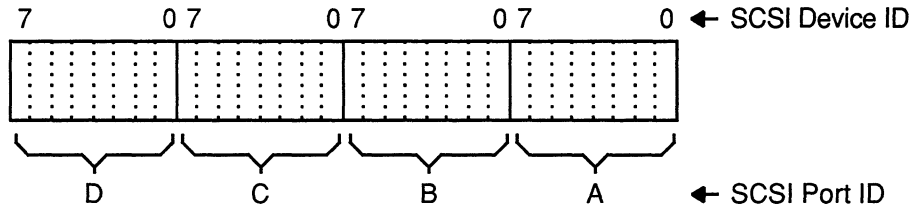
11.6.1 Disabling the Autoconfiguration of a SCSI Device

Note that, in special cases, you may need to prevent SYSGEN's autoconfiguration facility from loading the VMS disk or tape class driver for a device with a specific port ID and device ID. This would be the case if a non-Digital-supplied SCSI device should identify itself as either a random-access or sequential-access device and were to be controlled by the generic SCSI class driver.

To disable the loading of a VMS disk or tape driver for any given device ID, VMS Version 5.4 defines the special SYSGEN parameter **SCSI_NOAUTO**.

The **SCSI_NOAUTO** system parameter, as shown in Figure 11–3, stores a bit mask of 32 bits in which the low-order byte corresponds to the first SCSI bus (PKA0), the second byte corresponds to the second SCSI bus (PKB0), and so on. For each SCSI bus, setting the low-order bit inhibits automatic configuration of the device with SCSI device ID 0; setting the second low-order bit inhibits automatic configuration of the device with SCSI device ID 1, and so forth. For instance, the value 00002000_{16} would prevent the device with SCSI ID 5 on the bus identified by SCSI port ID *B* from being configured. By default, all of the bits in the mask are cleared, allowing all devices to be configured.

Figure 11–3 SCSI_NOAUTO System Parameter



11.7 Assigning a Channel to GKDRIVER

An application program assigns a channel to the generic SCSI class driver using the standard call to the $ASSIGN system service, as described in the *VMS System Services Reference Manual*. The application program specifies a device name and a word to receive the channel number.

11.8 Issuing a $QIO Request to the Generic Class Driver

The format of the Queue I/O Request ($QIO) system service that initiates a request to the SCSI generic class driver is as follows. This explanation concentrates on the special elements of a $QIO request to the SCSI generic class driver. For a detailed description of the $QIO system service, see the *VMS System Services Reference Manual*.

VAX MACRO Format

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

High-Level Language Format

```
SYS$QIO ([efn] ,chan ,func ,iosb ,[astadr] ,[astprm]
      ,p1 ,p2 [,p3] [,p4] [,p5] [,p6])
```

Arguments

chan

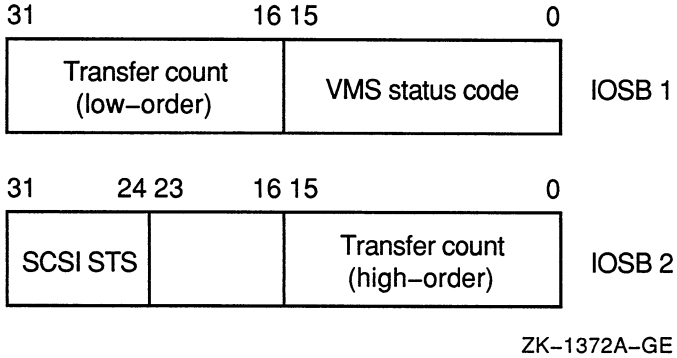
I/O channel assigned to the device to which the request is directed. The **chan** argument is a word value containing the number of the channel, as returned by the Assign I/O Channel ($ASSIGN) system service.

func

Longword value containing the IO$_DIAGNOSE function code. Only the IO$_DIAGNOSE function code is implemented in the generic SCSI class driver.

iosb

I/O status block. The **iosb** argument is required in a request to the generic SCSI class driver; it has the following format:

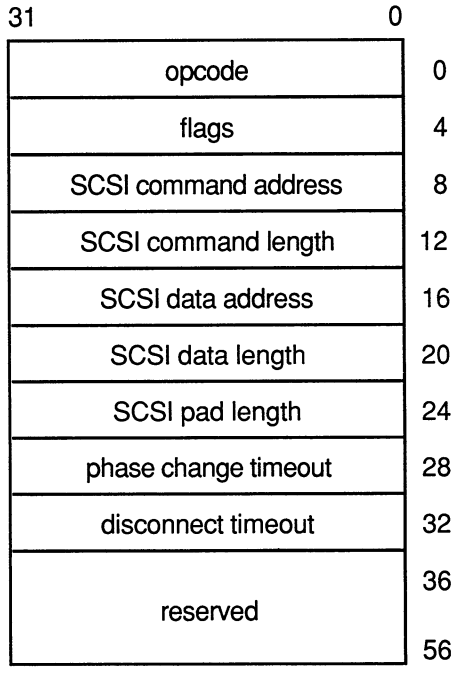


ZK–1372A–GE

The VMS status code provides the final status indicating the success or failure of the SCSI command. The SCSI status byte contains the status value returned from the target device, as defined in the ANSI SCSI specification. The transfer count field specifies the actual number of bytes transferred during the SCSI bus DATA IN or DATA OUT phase.

[efn]
[astadr]
[astprm]
These arguments apply to $QIO system service completion. For an explanation of these arguments, see the *VMS System Services Reference Manual*.

p1
Address of a generic SCSI descriptor of the following format:



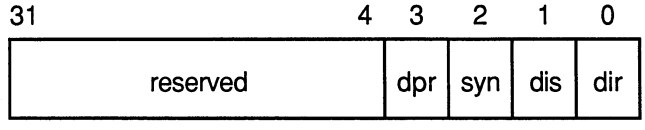
ZK–1373A–GE

p2
Length of the generic SCSI descriptor.

Descriptor Fields

opcode
Currently, the only supported opcode is 1, indicating a pass-through function. Other opcode values are reserved for future expansion.

flags
Bit map having the following format:



ZK–1374A–GE

Bits in the flags bit map are defined as follows:

Field	Definition
dir	Direction of transfer. If this bit is set, the target is expected at some time to enter the DATA IN phase to send data to the host. To facilitate this, the port driver maps the specified data buffer for write access. If this bit is clear, the target is expected at some time to enter the DATA OUT phase to receive data from the host. To facilitate this, the port driver maps the specified data buffer for read access. The generic SCSI class driver ignores the **dir** flag if either the **SCSI data address** or **SCSI data length** field of the generic SCSI descriptor is zero.
dis	Enable disconnection. If this bit is set, the target device is allowed to disconnect during the execution of the command. If this bit is clear, the target cannot disconnect during the execution of the command. Note that targets that hold on to the bus for long periods of time without disconnecting can adversely affect system performance. See Section 11.5.2 for additional information.
syn	Enable synchronous mode. If this bit is set, the port driver uses synchronous mode for data transfers, if both the host and target allow this mode of operation. If this bit is clear, or synchronous mode is not supported by either the host or target, the port driver uses asynchronous mode for data transfers. See Section 11.5.1 for additional information.
dpr	Disable port retry. If this bit is clear, the port driver retries, up to three times, any command that fails with a timeout, bus parity, or invalid phase transition error. If this bit is set, the port driver does not retry commands for which it detects failure. See Section 11.5.3 for additional information.

SCSI command address

Address of a buffer containing a SCSI command.

SCSI command length

Length of the SCSI command. The maximum length of the SCSI command is 128 bytes.

SCSI data address

Address of a data buffer associated with the SCSI command.

If the **dir** bit is set in the **flags** field, data is written into this buffer during the execution of the command. Otherwise, data is read from this buffer and sent to the target device.

If the SCSI command requires no data to be transferred, then the **SCSI data address** field should be clear.

SCSI data length
Length in bytes of the data buffer pointed to by the **SCSI data address** field. For the MicroVAX/VAXstation 3100 and VAXstation 3520/3540 systems, the maximum data buffer size is 65,535 bytes.

If the SCSI command requires no data to be transferred, then this field should be clear.

SCSI pad length
This field is used to accommodate SCSI device classes that require that the transfer length be specified in terms of a larger data unit than the count of bytes expressed in the **SCSI data length** field. If the total amount of data requested in the SCSI command does not match that specified in the **SCSI data length** field, this field must account for the difference.

For example, suppose an application program is using the generic class driver to read the first 2 bytes of a disk block. The length field in the SCSI READ command contains 1 (indicating one logical block, or 512 bytes), while the **SCSI data length** field contains a 2. The **SCSI pad length** field must contain the difference, 510 bytes.

For most transfers, this field should contain 0. Failure to initialize the **SCSI pad length** field properly causes port driver timeouts and SCSI bus resets.

phase change timeout
Maximum number of seconds for a target to change the SCSI bus phase or complete a data transfer. A value of 0 causes the SCSI port driver's default phase change timeout value of 4 seconds to be used.

See Section 11.5.4 for additional information.

disconnect timeout
Maximum number of seconds for a target to reselect the initiator to proceed with a disconnected I/O transfer. A value of 0 causes the SCSI port driver's default disconnect timeout value of 4 seconds to be used.

See Section 11.5.4 for additional information.

11.9 Generic SCSI Class Driver Device Information

A call to the Get Device/Volume Information ($GETDVI) system service returns the following information for any device serviced by the generic SCSI class driver. (See the description of the $GETDVI system service in the *VMS System Services Reference Manual*.)

$GETDVI returns the following device characteristics when you specify the item code DVI$_DEVCHAR:

DEV$M_AVL	Available device
DEV$M_IDV	Input device
DEV$M_ODV	Output device
DEV$M_SHR	Shareable device
DEV$M_RND	Random-access device

DVI$DEVCLASS returns the device class, which is DC$_MISC; DVI$DEVTYPE returns the device type, which is DT$_GENERIC_SCSI.

11.10 Generic SCSI Class Driver Programming Example

The following application program uses the generic SCSI class driver to send a SCSI INQUIRY command to a device.

```
/*
GKTEST.C

This program uses the SCSI generic class driver to send an inquiry command
to a device on the SCSI bus.

*/

#include ctype

/* Define the descriptor used to pass the SCSI information to GKDRIVER */

#define OPCODE 0
#define FLAGS 1
#define COMMAND_ADDRESS 2
#define COMMAND_LENGTH 3
#define DATA_ADDRESS 4
#define DATA_LENGTH 5
#define PAD_LENGTH 6
#define PHASE_TIMEOUT 7
#define DISCONNECT_TIMEOUT 8

#define FLAGS_READ 1
#define FLAGS_DISCONNECT 2

#define GK_EFN 1

#define SCSI_STATUS_MASK 0X3E

#define INQUIRY_OPCODE 0x12
#define INQUIRY_DATA_LENGTH 0x30

globalvalue
        IO$_DIAGNOSE;

short
        gk_chan,
        transfer_length;

int
        i,
        status,
        gk_device_desc[2],
        gk_iosb[2],
        gk_desc[15];

char
        scsi_status,
        inquiry_command[6] = {INQUIRY_OPCODE, 0, 0, 0, INQUIRY_DATA_LENGTH, 0},
        inquiry_data[INQUIRY_DATA_LENGTH],
        gk_device[] = {"GKA0"};

main ()
{

/* Assign a channel to GKA0 */

        gk_device_desc[0] = 4;
        gk_device_desc[1] = &gk_device[0];
        status = sys$assign (&gk_device_desc[0], &gk_chan, 0, 0);
        if (!(status & 1)) sys$exit (status);

/* Set up the descriptor with the SCSI information to be sent to the target */
```

```
        gk_desc[OPCODE] = 1;
        gk_desc[FLAGS] = FLAGS_READ + FLAGS_DISCONNECT;
        gk_desc[COMMAND_ADDRESS] = &inquiry_command[0];
        gk_desc[COMMAND_LENGTH] = 6;
        gk_desc[DATA_ADDRESS] = &inquiry_data[0];
        gk_desc[DATA_LENGTH] = INQUIRY_DATA_LENGTH;
        gk_desc[PAD_LENGTH] = 0;
        gk_desc[PHASE_TIMEOUT] = 0;
        gk_desc[DISCONNECT_TIMEOUT] = 0;
        for (i=9; i<15; i++) gk_desc[i] = 0;    /* Clear reserved fields */

/* Issue the QIO to send the inquiry command and receive the inquiry data */

        status = sys$qiow (GK_EFN, gk_chan, IO$_DIAGNOSE, gk_iosb, 0, 0,
                           &gk_desc[0], 15*4, 0, 0, 0, 0);

/* Check the various returned status values */

        if (!(status & 1)) sys$exit (status);
        if (!(gk_iosb[0] & 1)) sys$exit (gk_iosb[0] & 0xffff);
        scsi_status = (gk_iosb[1] >> 24) & SCSI_STATUS_MASK;
        if (scsi_status) {
                printf ("Bad SCSI status returned: %02.2x\n", scsi_status);
                sys$exit (1);
        }

/* The command succeeded. Display the SCSI data returned from the target */

        transfer_length = gk_iosb[0] >> 16;
        printf ("SCSI inquiry data returned: ");
        for (i=0; i<transfer_length; i++) {
                if (isprint (inquiry_data[i]))
                        printf ("%c", inquiry_data[i]);
                else
                        printf (".");
        }
        printf ("\n");
}
```

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 ACP-QIO 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)	(none)

[1]Only for IO$_CREATE and IO$_ACCESS

[2]Only for IO$_CREATE and IO$_DELETE

[3]Only for IO$_ACPCONTROL

QIO Status Returns		
SS$_ACCONFLICT	SS$_ACPVAFUL	SS$_BADATTRIB
SS$_BADCHKSUM	SS$_BADFILEHDR	SS$_BADFILENAME
SS$_BADFILEVER	SS$_BADIRECTORY	SS$_BADPARAM
SS$_BADQFILE	SS$_BLOCKCNTERR	SS$_CREATED
SS$_DEVICEFULL	SS$_DIRFULL	SS$_DIRNOTEMPTY
SS$_DUPDSKQUOTA	SS$_DUPFILENAME	SS$_ENDOFFILE
SS$_EXBYTLM	SS$_EXDISKQUOTA	SS$_FCPREADERR
SS$_FCPREWNDERR	SS$_FCPSPACERR	SS$_FCPWRITERR
SS$_FILELOCKED	SS$_FILENUMCHK	SS$_FILEPURGED
SS$_FILESEQCHK	SS$_FILESTRUCT	SS$_FILNOTEXP
SS$_HEADERFULL	SS$_IBCERROR[1]	SS$_IDXFILEFULL
SS$_ILLCNTRFUNC	SS$_NODISKQUOTA	SS$_NOMOREFILES
SS$_NOPRIV	SS$_NOQFILE	SS$_NOSUCHFILE

[1]The second longword of the IOSB contains a job controller status code.

I/O Function Codes

A.1 ACP-QIO Interface Driver

QIO Status Returns		
SS$_NOTAPEOP	SS$_NOTLABELMT	SS$_NOTPRINTED[1]
SS$_NOTVOLSET	SS$_OVRDSKQUOTA	SS$_QFACTIVE
SS$_QFNOTACT	SS$_SERIOUSEXCP	SS$_SUPERSEDE
SS$_TAPEPOSLOST	SS$_TOOMANYVER	SS$_WRITLCK
SS$_WRONGACP		

[1]The second longword of the IOSB contains a job controller status code.

A.2 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)	(none)

QIO Status Returns			
SS$_ABORT	SS$_DATAOVERUN	SS$_ENDOFFILE	SS$_NORMAL

A.3 Disk Drivers

Functions	Arguments	Modifiers
IO$_READVBLK IO$_READLBLK IO$_READPBLK[4] IO$_WRITEVBLK IO$_WRITELBLK IO$_WRITEPBLK[4]	P1 - buffer address P2 - byte count P3 - disk address	IO$M_INHSEEK[1] IO$M_DATACHECK[2] IO$M_DELDATA[3] IO$M_INHRETRY IO$M_ERASE[5]
IO$_WRITECHECK[2]	P1 - buffer address P2 - byte count P3 - disk address	(none)

[1]Only for IO$READPBLK and IO$_WRITEPBLK (not for TU58, RX01, RX02, RB02, or RL02)

[2]Not for RX01 and RX02

[3]Only for IO$_WRITEPBLK on RX02

[4]Not for DSA disks

[5]Only for write functions

Functions	Arguments	Modifiers
IO$_SENSECHAR IO$_SENSEMODE IO$_PACKACK IO$_AVAILABLE IO$_UNLOAD	(none)	(none)
IO$_SEARCH	P1 - read/write head position	(none)
IO$_SEEK[4]	P1 - seek to specified cylinder	(none)
IO$_FORMAT	P1 - RX02 density	(none)
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$M_CREATE[6] IO$M_ACCESS[6] IO$M_DELETE[7] IO$M_DMOUNT[8]

[4]Not for DSA disks

[6]Only for IO$_CREATE and IO$_ACCESS

[7]Only for IO$_CREATE and IO$_DELETE

[8]Only for IO$_ACPCONTROL

QIO Status Returns		
SS$_ABORT	SS$_CANCEL	SS$_CTRLERR
SS$_DATACHECK	SS$_DATAOVERUN	SS$_DRVERR
SS$_FORCEDERR	SS$_FORMAT	SS$_ILLIOFUNC
SS$_IVADDR	SS$_IVBUFLEN	SS$_MEDOFL
SS$_NONEXDRV	SS$_NORMAL	SS$_OPINCOMPL
SS$_PARITY	SS$_RCT	SS$_RDDELDATA
SS$_TIMEOUT	SS$_UNSAFE	SS$_VOLINV
SS$_WASECC	SS$_WRITLCK	

A.4 Laboratory Peripheral Accelerator Driver

Functions	Arguments	Modifiers
IO$_LOADMCODE	P1 - starting address of microcode to be loaded P2 - load byte count P3 - starting microprogram address to receive microcode	(none)
IO$_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 (two's complement)	(none)
IO$_STARTDATA	P1 - data transfer command table address P2 - data transfer command table length P3 - normal completion AST address P4 - overrun completion AST address	IO$_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 Clock B rate.
LPA$XRATE	Compute clock rate and preset value.
LPA$IBFSTS	Return buffer status.
LPA$IGTBUF	Return next available buffer.

High-Level Language Subroutines	Functions
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 LPA11-K.

QIO Status Returns		
SS$_ABORT	SS$_BADPARAM	SS$_BUFNOTALIGN
SS$_CANCEL	SS$_CTRLERR	SS$_DATACHECK
SS$_DEVACTIVE	SS$_DEVCMDERR	SS$_DEVREQERR
SS$_EXQUOTA	SS$_INSFBUFDP	SS$_INSFMAPREQ
SS$_INSFMEM	SS$_IVBUFLEN	SS$_IVMODE
SS$_MCNOTVALID	SS$_PARITY	SS$_POWERFAIL
SS$_TIMEOUT		

A.5 Line Printer Driver

Functions	Arguments	Modifiers
IO$_WRITEVBLK IO$_WRITELBLK IO$_WRITEPBLK	P1 - buffer address P2 - buffer size P3 - (ignored) P4 - carriage control specifier[1]	(none)
IO$_SENSEMODE	(none)	(none)
IO$_SETMODE IO$_SETCHAR	P1 - characteristics buffer address	(none)

[1]Only for IO$_WRITEVBLK and IO$_WRITELBLK

QIO Status Returns			
SS$_ABORT	SS$_ACCVIO	SS$_CANCEL	SS$_NORMAL

A.6 Magnetic Tape Drivers

Functions	Arguments	Modifiers
IO$_READVBLK IO$_READLBLK IO$_READPBLK	P1 - buffer address P2 - byte count	IO$M_DATACHECK[1] IO$M_INHRETRY IO$M_REVERSE[3]
IO$_WRITEVBLK IO$_WRITELBLK IO$_WRITEPBLK	P1 - buffer address P2 - byte count	IO$M_DATACHECK[1] IO$M_INHRETRY IO$M_INHEXTGAP[2] IO$M_NOWAIT[8] IO$M_ERASE[7]
IO$_SETMODE IO$_SETCHAR	P1 - characteristics buffer address P2 - characteristics buffer length[9]	
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[4] IO$M_ACCESS[4] IO$M_DMOUNT[5]
IO$_SKIPFILE	P1 - skip n tape marks	IO$M_INHRETRY IO$M_NOWAIT[8]
IO$_SKIPRECORD	P1 - skip n blocks	IO$M_INHRETRY IO$M_NOWAIT[8]
IO$_REWIND IO$_REWINDOFF IO$_UNLOAD	(none)	IO$M_INHRETRY IO$M_NOWAIT
IO$_WRITEOF	(none)	IO$M_INHEXTGAP[2] IO$M_INHRETRY IO$M_NOWAIT[8]
IO$_SENSEMODE IO$_SENSECHAR	P1 - characteristics buffer address[9] P2 - characteristics buffer length[9]	IO$M_INHRETRY

[1]Not for TS04 and TU80

[2]Only for TE16, TU45, and TU77

[3]Not for TK50

[4]Only for IO$_CREATE and IO$_ACCESS

[5]Only for IO$_ACPCONTROL

[7]IO$M_ERASE takes no arguments; only for IO$_WRITELBLK and IO$_WRITEPBLK on TMSCP drives.

[8]Only for TU81-Plus drives

[9]Only for TMSCP drives

Functions	Arguments	Modifiers
IO$_DSE[6] IO$_PACKACK IO$_AVAILABLE	(none)	(none)

[6]Only for TU78, TU81, TA81, and TA78

QIO Status Returns		
SS$_ABORT	SS$_CANCEL	SS$_CTRLERR
SS$_DATACHECK	SS$_DATAOVERUN	SS$_DEVOFFLINE
SS$_DRVERR	SS$_ENDOFFILE	SS$_ENDOFTAPE
SS$_ENDOFVOLUME	SS$_FORMAT	SS$_ILLIOFUNC
SS$_MEDOFL	SS$_NONEXDRV	SS$_NORMAL
SS$_OPINCOMPL	SS$_PARITY	SS$_SERIOUSEXCP
SS$_TIMEOUT	SS$_UNSAFE	SS$_VOLINV
SS$_WRITLCK		

A.7 Mailbox Driver

Functions	Arguments	Modifiers
IO$_READVBLK IO$_READLBLK IO$_READPBLK IO$_WRITEVBLK IO$_WRITELBLK IO$_WRITEPBLK	P1 - buffer address P2 - buffer size	IO$M_NOW IO$M_NORSWAIT[1]
IO$_WRITEOF	(none)	IO$M_NOW
IO$_SETMODE!IO$M_READATTN IO$_SETMODE!IO$M_WRTATTN	P1 - AST address P2 - AST parameter P3 - access mode	(none)
IO$_SETMODE!IO$M_SETPROT	P2 - volume protection mask	(none)

[1]Only for write functions

QIO Status Returns			
SS$_ABORT	SS$_BUFFEROVF	SS$_ENDOFFILE	SS$_NORMAL

A.8 Terminal Driver

Functions	Arguments	Modifiers
IO$_READVBLK IO$_READLBLK IO$_READPROMPT	P1 - buffer address P2 - buffer size P3 - timeout P4 - read terminator block address P5 - prompt string buffer address P6 - prompt string buffer size[1]	IO$M_NOECHO IO$M_CVTLOW IO$M_NOFILTR IO$M_TIMED IO$M_PURGE IO$M_DSABLMBX IO$M_TRMNOECHO IO$M_ESCAPE
IO$_READVBLK	P1 - buffer address P2 - buffer size P3 - access mode to probe itemlist P4 - (zero) P5 - itemlist buffer address P6 - itemlist buffer size	IO$M_EXTEND[2]
IO$_WRITEVBLK IO$_WRITELBLK IO$_WRITEPBLK	P1 - buffer address P2 - buffer size P3 - (ignored) P4 - carriage control specifier[3]	IO$M_CANCTRLO IO$M_ENABLMBX IO$M_NOFORMAT IO$M_REFRESH IO$M_BREAKTHRU
IO$_SETMODE IO$_SETCHAR	P1 - characteristics buffer address P2 - characteristics buffer size P3 - speed specifier P4 - fill specifier P5 - parity flags	
IO$_SETMODE IO$_SETCHAR	(none)	IO$M_HANGUP
IO$_SETMODE	P1 - buffer address P2 - buffer size	IO$M_BRDCST
IO$_SETMODE IO$_SETCHAR	P1 - AST service routine address P2 - AST parameter P3 - access mode to deliver AST	IO$M_CTRLCAST IO$M_CTRLYAST

[1]Only for IO$_READPROMPT

[2]Only for itemlist read function. Do not specify with other modifiers.

[3]Only for IO$_WRITELBLK and IO$_WRITEVBLK

Functions	Arguments	Modifiers
IO$_SETMODE IO$_SETCHAR	P1 - AST service routine address P2 - character mask address P3 - access mode to deliver AST	IO$M_OUTBAND IO$M_TT_ABORT[4] IO$M_INCLUDE[4]
IO$_SETMODE IO$_SETCHAR	P1 - address of control signals	IO$M_SET_MODEM[5] IO$M_MAINT
IO$_SETMODE IO$_SETCHAR	(none)	IO$M_LOOP[5] IO$M_UNLOOP[5] IO$M_MAINT
IO$_TTY_PORT		IO$M_LT_CONNECT IO$M_LT_DISCON
IO$_TTY_PORT	P1 - item list[6] address P2 - queued status	IO$M_LT_MAP_PORT
IO$_TTY_PORT	P1 - service name descriptor address P2 - service rating	IO$M_LT_RATING
IO$_SENSEMODE IO$_SENSECHAR	P1 - characteristics buffer address P2 - characteristics buffer size	IO$M_TYPEAHDCNT
IO$_SENSEMODE IO$_SENSECHAR	P1 - address of input modem signal block	IO$M_RD_MODEM
IO$_SENSEMODE	P1 - buffer address P2 - buffer size	IO$M_BRDCST

[4]Only with IO$M_OUTBAND

[5]Only with IO$M_MAINT

[6]Item list: IO$V_LT_MAP_NODNAM, IO$V_LT_MAP_PORNAM, IO$V_LT_MAP_SRVNAM, IO$V_LT_MAP_LNKNAM, and IO$V_LT_MAP_NETADR.

QIO Status Returns		
SS$_ABORT	SS$_BADESCAPE	SS$_BADPARAM
SS$_CANCEL	SS$_CONTROLC	SS$_CONTROLO
SS$_CONTROLY	SS$_DATAOVERUN	SS$_INCOMPAT
SS$_NORMAL	SS$_PARITY	SS$_PARTESCAPE
SS$_TIMEOUT		

B Tables

Table B–1 lists the DEC Multinational Character Set. The DEC Multinational Character set is an eight-bit character set with 256 characters. The first 128 characters in the set correspond to the ASCII character set. The *VAX EDT Reference Manual* lists the graphics for these characters and describes how to enter them from various types of terminals.

Table B–1 DEC Multinational Character Set

Hex Code	Octal Code	Decimal Code	Char or Abbrev.	Description
		ASCII Control Characters[1]		
00	000	000	NUL	null character
01	001	001	SOH	start of heading (CTRL/A)
02	002	002	STX	start of text (CTRL/B)
03	003	003	ETX	end of text (CTRL/C)
04	004	004	EOT	end of transmission (CTRL/D)
05	005	005	ENQ	enquiry (CTRL/E)
06	006	006	ACK	acknowledge (CTRL/F)
07	007	007	BEL	bell (CTRL/G)
08	010	008	BS	backspace (CTRL/H)
09	011	009	HT	horizontal tabulation (CTRL/I)
0A	012	010	LF	line feed (CTRL/J)
0B	013	011	VT	vertical tabulation (CTRL/K)
0C	014	012	FF	form feed (CTRL/L)
0D	015	013	CR	carriage return (CTRL/M)
0E	016	014	SO	shift out (CTRL/N)
0F	017	015	SI	shift in (CTRL/O)
10	020	016	DLE	data link escape (CTRL/P)
11	021	017	DC1	device control 1 (CTRL/Q)
12	022	018	DC2	device control 2 (CTRL/R)
13	023	019	DC3	device control 3 (CTRL/S)
14	024	020	DC4	device control 4 (CTRL/T)
15	025	021	NAK	negative acknowlege (CTRL/U)

[1]The ALTMODE and DELETE characters (decimal 125, 126, and 127) are also control characters.

(continued on next page)

Table B–1 (Cont.) DEC Multinational Character Set

Hex Code	Octal Code	Decimal Code	Char or Abbrev.	Description
		ASCII Control Characters[1]		
16	026	022	SYN	synchronous idle (CTRL/V)
17	027	023	ETB	end of transmission block (CTRL/W)
18	030	024	CAN	cancel (CTRL/X)
19	031	025	EM	end of medium (CTRL/Y)
1A	032	026	SUB	substitute (CTRL/Z)
1B	033	027	ESC	escape
1C	034	028	FS	file separator
1D	035	029	GS	group separator
1E	036	030	RS	record separator
1F	037	031	US	unit separator
		ASCII Special and Numeric Characters		
20	040	032	SP	space
21	041	033	!	exclamation point
22	042	034	"	quotation marks (double quote)
23	043	035	#	number sign
24	044	036	$	dollar sign
25	045	037	%	percent sign
26	046	038	&	ampersand
27	047	039	'	apostrophe (single quote)
28	050	040	(	opening parenthesis
29	051	041	)	closing parenthesis
2A	052	042	*	asterisk
2B	053	043	+	plus
2C	054	044	,	comma
2D	055	045	–	hyphen or minus
2E	056	046	.	period or decimal point
2F	057	047	/	slash
30	060	048	0	zero
31	061	049	1	one
32	062	050	2	two
33	063	051	3	three

[1]The ALTMODE and DELETE characters (decimal 125, 126, and 127) are also control characters.

(continued on next page)

Table B–1 (Cont.) DEC Multinational Character Set

Hex Code	Octal Code	Decimal Code	Char or Abbrev.	Description
		ASCII Special and Numeric Characters		
34	064	052	4	four
35	065	053	5	five
36	066	054	6	six
37	067	055	7	seven
38	070	056	8	eight
39	071	057	9	nine
3A	072	058	:	colon
3B	073	059	;	semicolon
3C	074	060	<	less than
3D	075	061	=	equals
3E	076	062	>	greater than
3F	077	063	?	question mark
		ASCII Alpha Characters		
40	100	064	@	commercial at sign
41	101	065	A	uppercase A
42	102	066	B	uppercase B
43	103	067	C	uppercase C
44	104	068	D	uppercase D
45	105	069	E	uppercase E
46	106	070	F	uppercase F
47	107	071	G	uppercase G
48	110	072	H	uppercase H
49	111	073	I	uppercase I
4A	112	074	J	uppercase J
4B	113	075	K	uppercase K
4C	114	076	L	uppercase L
4D	115	077	M	uppercase M
4E	116	078	N	uppercase N
4F	117	079	O	uppercase O
50	120	080	P	uppercase P
51	121	081	Q	uppercase Q
52	122	082	R	uppercase R
53	123	083	S	uppercase S

(continued on next page)

Table B–1 (Cont.) DEC Multinational Character Set

Hex Code	Octal Code	Decimal Code	Char or Abbrev.	Description
		ASCII Alpha Characters		
54	124	084	T	uppercase T
55	125	085	U	uppercase U
56	126	086	V	uppercase V
57	127	087	W	uppercase W
58	130	088	X	uppercase X
59	131	089	Y	uppercase Y
5A	132	090	Z	uppercase Z
5B	133	091	[	left bracket
5C	134	092	\	backslash
5D	135	093	]	right bracket
5E	136	094	^	circumflex
5F	137	095	_	underscore
60	140	096	‘	grave accent
61	141	097	a	lowercase a
62	142	098	b	lowercase b
63	143	099	c	lowercase c
64	144	100	d	lowercase d
65	145	101	e	lowercase e
66	146	102	f	lowercase f
67	147	103	g	lowercase g
68	150	104	h	lowercase h
69	151	105	i	lowercase i
6A	152	106	j	lowercase j
6B	153	107	k	lowercase k
6C	154	108	l	lowercase l
6D	155	109	m	lowercase m
6E	156	110	n	lowercase n
6F	157	111	o	lowercase o
70	160	112	p	lowercase p
71	161	113	q	lowercase q
72	162	114	r	lowercase r
73	163	115	s	lowercase s
74	164	116	t	lowercase t
75	165	117	u	lowercase u

(continued on next page)

Table B–1 (Cont.) DEC Multinational Character Set

Hex Code	Octal Code	Decimal Code	Char or Abbrev.	Description
		ASCII Alpha Characters		
76	166	118	v	lowercase v
77	167	119	w	lowercase w
78	170	120	x	lowercase x
79	171	121	y	lowercase y
7A	172	122	z	lowercase z
7B	173	123	{	left brace
7C	174	124	|	vertical line
7D	175	125	}	right brace (ALTMODE)
7E	176	126	~	tilde (ALTMODE)
7F	177	127	DEL	rubout (DELETE)
80	200	128	—	[reserved]
81	201	129	—	[reserved]
82	202	130	—	[reserved]
83	203	131	—	[reserved]
84	204	132	IND	index
85	205	133	NEL	next line
86	206	134	SSA	start of selected area
87	207	135	ESA	end of started area
88	210	136	HTS	horizontal tab set
89	211	137	HTJ	horizontal tab set with justification
8A	212	138	VTS	vertical tab set
8B	213	139	PLD	partial line down
8C	214	140	PLU	partial line up
8D	215	141	RI	reverse index
8E	216	142	SS2	single shift 2
8F	217	143	SS3	single shift 3
90	220	144	DCS	device control string
91	221	145	PU1	private use 1
92	222	146	PU2	private use 2
93	223	147	STS	set transmit state
94	224	148	CCH	cancel character
95	225	149	MW	message waiting
96	226	150	SPA	start of protected area
97	227	151	EPA	end of protected area

(continued on next page)

Table B–1 (Cont.) DEC Multinational Character Set

Hex Code	Octal Code	Decimal Code	Char or Abbrev.	Description
		ASCII Alpha Characters		
98	230	152	—	[reserved]
99	231	153	—	[reserved]
9A	232	154	—	[reserved]
9B	233	155	CSI	control sequence introducer
9C	234	156	ST	string terminator
9D	235	157	OSC	operating system command
9E	236	158	PM	privacy message
9F	237	159	APC	application
A0	240	160	—	[reserved]
A1	241	161	¡	inverted exclamation point
A2	242	162	¢	cent sign
A3	243	163	£	pound sign
A4	244	164	—	[reserved]
A5	245	165	¥	yen sign
A6	246	166	—	[reserved]
A7	247	167	§	section sign
A8	250	168	¤	general currency sign
A9	251	169	©	copyright sign
AA	252	170	ª	feminine ordinal indicator
AB	253	171	«	angle quotation mark left
AC	254	172	—	[reserved]
AD	255	173	—	[reserved]
AE	256	174	—	[reserved]
AF	257	175	—	[reserved]
B0	260	176	°	degree sign
B1	261	177	±	plus/minus sign
B2	262	178	²	superscript 2
B3	263	179	³	superscript 3
B4	264	180	—	[reserved]
B5	265	181	µ	micro sign
B6	266	182	¶	paragraph sign, pilcrow
B7	267	183	·	middle dot
B8	270	184	—	[reserved]
B9	271	185	¹	superscript 1

(continued on next page)

Table B–1 (Cont.) DEC Multinational Character Set

Hex Code	Octal Code	Decimal Code	Char or Abbrev.	Description
		ASCII Alpha Characters		
BA	272	186	º	masculine ordinal indicator
BB	273	187	»	angle quotation mark right
BC	274	188	¼	fraction one-quarter
BD	275	189	½	fraction one-half
BE	276	190	—	[reserved]
BF	277	191	¿	inverted question mark
C0	300	192	À	uppercase A with grave accent
C1	301	193	Á	uppercase A with acute accent
C2	302	194	Â	uppercase A with circumflex
C3	303	195	Ã	uppercase A with tilde
C4	304	196	Ä	uppercase A with umlaut, (diaeresis)
C5	305	197	Å	uppercase A with ring
C6	306	198	Æ	uppercase AE diphthong
C7	307	199	Ç	uppercase C with cedilla
C8	310	200	È	uppercase E with grave accent
C9	311	201	É	uppercase E with acute accent
CA	312	202	Ê	uppercase E with circumflex
CB	313	203	Ë	uppercase E with umlaut, (diaeresis)
CC	314	204	Ì	uppercase I with grave accent
CD	315	205	Í	uppercase I with acute accent
CE	316	206	Î	uppercase I with circumflex
CF	317	207	Ï	uppercase I with umlaut, (diaeresis)
D0	320	208	—	[reserved]
D1	321	209	Ñ	uppercase N with tilde
D2	322	210	Ò	uppercase O with grave accent
D3	323	211	Ó	uppercase O with acute accent

(continued on next page)

Table B–1 (Cont.) DEC Multinational Character Set

Hex Code	Octal Code	Decimal Code	Char or Abbrev.	Description
		ASCII Alpha Characters		
D4	324	212	Ô	uppercase O with circumflex
D5	325	213	Õ	uppercase O with tilde
D6	326	214	Ö	uppercase O with umlaut, (diaeresis)
D7	327	215	Œ	uppercase OE ligature
D8	330	216	Ø	uppercase O with slash
D9	331	217	Ù	uppercase U with grave accent
DA	332	218	Ú	uppercase U with acute accent
DB	333	219	Û	uppercase U with circumflex
DC	334	220	Ü	uppercase U with umlaut, (diaeresis)
DD	335	221	Ÿ	uppercase Y with umlaut, (diaeresis)
DE	336	222	—	[reserved]
DF	337	223	ß	German lowercase sharp s
E0	340	224	à	lowercase a with grave accent
E1	341	225	á	lowercase a with acute accent
E2	342	226	â	lowercase a with circumflex
E3	343	227	ã	lowercase a with tilde
E4	344	228	ä	lowercase a with umlaut, (diaeresis)
E5	345	229	å	lowercase a with ring
E6	346	230	æ	lowercase ae diphthong
E7	347	231	ç	lowercase c with cedilla
E8	350	232	è	lowercase e with grave accent
E9	351	233	é	lowercase e with acute accent
EA	352	234	ê	lowercase e with circumflex
EB	353	235	ë	lowercase e with umlaut, (diaeresis)
EC	354	236	ì	lowercase i with grave accent

(continued on next page)

Table B–1 (Cont.) DEC Multinational Character Set

Hex Code	Octal Code	Decimal Code	Char or Abbrev.	Description
		ASCII Alpha Characters		
ED	355	237	í	lowercase i with acute accent
EE	356	238	î	lowercase i with circumflex
EF	357	239	ï	lowercase i with umlaut, (diaeresis)
F0	360	240	—	[reserved]
F1	361	241	ñ	lowercase n with tilde
F2	362	242	ò	lowercase o with grave accent
F3	363	243	ó	lowercase o with acute accent
F4	364	244	ô	lowercase o with circumflex
F5	365	245	õ	lowercase o with tilde
F6	366	246	ö	lowercase o with umlaut, (diaeresis)
F7	367	247	œ	lowercase oe ligature
F8	370	248	ø	lowercase o with slash
F9	371	249	ù	lowercase u with grave accent
FA	372	250	ú	lowercase u with acute accent
FB	373	251	û	lowercase u with circumflex
FC	374	252	ü	lowercase u with umlaut, (diaeresis)
FD	375	253	ÿ	lowercase y with umlaut, (diaeresis)
FE	376	254	—	[reserved]
FF	377	255	—	[reserved]

B.1 Terminal Sequences and Modes

Table B–2 lists the valid ANSI and Digital-private escape sequences for terminals that have the TT2$M_ANSICRT, TT2$M_DECCRT, TT2$M_AVO, TT2$M_EDIT, and TT2$M_BLOCK characteristics (see Section 8). Table B–2 also lists assumed and selectable ANSI modes and selectable Digital-private modes. Only the names of the escape sequences and modes are listed (for more information see the specific VT100-, VT200-, or VT300- family user's guide). Unless otherwise noted, the operation of escape sequences and modes is identical to the particular VT100-, VT200-, or VT300- family terminals that implement these features.

Tables

B.1 Terminal Sequences and Modes

Table B–2 Sequences and Modes

Name	Valid Parameters	ANSICRT	DECCRT	AVO	EDIT	BLOCK[1]
		ANSI-Defined Escape Sequences				
CPR	All	x	x			
CUB	All	x	x			
CUD	All	x	x			
CUF	All	x	x			
CUP	All	x	x			
CUU	All	x	x			
DSR	0,3,5,6	x	x			
ED	0,1,2	x	x			
EL	0,1,2	x	x			
HVP	All	x	x			
IND		x	x			
NEL		x	x			
RI		x	x			
RIS		x	x			
SCS	UK,ASCII,0	x				
SCS	UK,ASCII	x	x			
SGR	0,4,7	x	x			
SGR	0,1,4,5,7			x		
DA	Terminal specific			x		
HTS			x			
RM	Class specific		x			
SM	Class specific		x			
TBC	0,3		x			
DCH	All				x	x
DL	All				x	x
IL	All				x	x
		Digital-Private Escape Sequences				
DECDHDL	2,3		x			
DECDWL	6		x			
DECKPAM			x			
DECKPNM			x			
DECRC	8		x			
DECSC	7		x			

[1]Terminal characteristics. Prefix is TT2$M_.

(continued on next page)

Table B–2 (Cont.) Sequences and Modes

Name	Valid Parameters	ANSICRT	DECCRT	AVO	EDIT	BLOCK[1]
			Digital-Private Escape Sequences			
DECSTBM	All		x			
DECSWL	5		x			
DECPRO	0,1,4,5,7,254					x
DECTTC	0,1					x
DECXMIT	5					x
			ANSI Selectable Modes (Set with ANSI SM/RM)			
IRM	4				x	x
GATM	1				x	x
ERM	6					x
TTM	16					x
			Digital-Private Selectable Modes (Set with ANSI SM/RM)			
DECCKM	1			x		
DECANM	2			x		
DECCOLM	3			x		
DECSCLM	4			x		
DECSCNM	5			x		
DECOM	6			x		
DECAWM	7			x		
DECARM	8			x		
DECEDM	10					x
DECEKEM	16					x
DECLTM	11					x
DECSCFDM	13					x
DECTEM	14					x
			ANSI Assumed Modes			
CRM		Reset	Reset			
EBM		Reset	Reset			
ERM		Set	Set		[2]	
FEAM		Reset	Reset			

[1]Terminal characteristics. Prefix is TT2$M_.

[2]Selectable mode.

(continued on next page)

Table B–2 (Cont.) Sequences and Modes

Name	Valid Parameters	ANSICRT	DECCRT	AVO	EDIT	BLOCK[1]
		ANSI Assumed Modes				
FETM		Reset	Reset			
GATM		N/A	N/A		[2]	
HEM		N/A	N/A			
IRM		Reset	Reset	[2]	[2]	
KAM		Reset	Reset			
MATH		N/A	N/A			
PUM		Reset	Reset			
SATM		N/A	N/A			
SRTM		Reset	Reset			
TSM		Reset	Reset			
TTM		N/A	N/A		[2]	
VEM		N/A	N/A			

[1]Terminal characteristics. Prefix is TT2$M_.

[2]Selectable mode.

C Control Connection Routines

This appendix lists and describes the VAX calling standards for the pseudoterminal driver control connection routines. The routines appear in this section in alphabetical order. Table C–1 lists the control connection routines and their functions.

Table C–1 Control Connection Routines

Routine Name	Description
PTD$CANCEL	Cancels a queued control connection read request
PTD$CREATE	Creates a pseudoterminal
PTD$DELETE	Deletes a pseudoterminal
PTD$READ	Reads data from the pseudoterminal
PTD$SET_EVENT_NOTIFICATION	Enables or disables terminal event notification ASTs
PTD$WRITE	Writes data to the pseudoterminal

PTD$CANCEL Cancel Queued Request

Cancels a queued control connection read request.

FORMAT

PTD$CANCEL *chan*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

chan

VMS usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Number of the I/O channel assigned to the pseudoterminal.

RETURN VALUES

SS$_NORMAL	Normal successful completion.
SS$_DEVOFFLINE	Device is off line and request cannot proceed.
SS$_IVCHAN	Illegal channel.
SS$_NOPRIV	Insufficient privilege to perform request.

PTD$CREATE Create a Pseudoterminal

Creates a new pseudoterminal with a unique device name.

FORMAT

PTD$CREATE *chan [,acmode] [,charbuff] [,buflen] [,astadr] [astprm] [,ast_acmode], inadr*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

chan

VMS usage: **channel**
type: **word (unsigned)**
access: **write only**
mechanism: **by reference**

Number of the channel that is assigned to the new pseudoterminal. This argument is the address of a word into which PTD$CREATE writes the channel number.

acmode

VMS usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode to be associated with the channel. The most privileged access mode is the access mode of the caller. I/O operations on the channel can be performed only from equal and more privileged access modes.

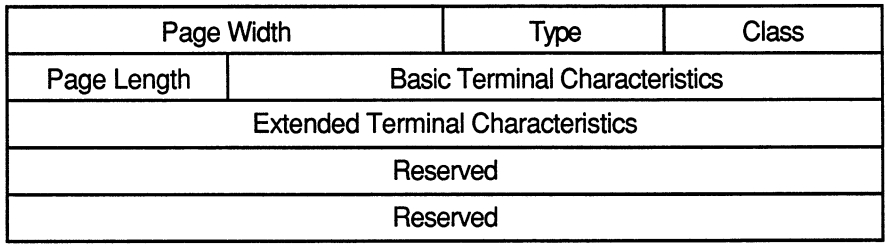
charbuff

VMS usage: **device_characteristics**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Address of buffer containing the device characteristics. This information is used to set up the pseudoterminal's initial characteristics. This buffer can be 12, 16, or 20 bytes long.

Figure C–1 shows the format of this buffer.

Figure C–1 Device Characteristics Buffer



ZK–9573–GE

buflen

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Length of the characteristics buffer (either 12, 16, or 20 bytes). This argument is required if you supply the **charbuff** argument.

astadr

VMS usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed when the terminal connection deassigns the last channel to the pseudoterminal. This argument is the address of the entry mask of this routine. This is a repeating AST and is active until the control connection deletes the pseudoterminal.

astprm

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter to be passed to the AST service routine specified by **astadr**.

ast_acmode

VMS usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode for which the AST is to be declared. The most privileged access mode is the access mode of the caller. The resulting mode is the access mode at which the AST is declared.

inadr

VMS usage: **address_range**
type: **longword (unsigned)**
access: **read only**
mechanism: **by reference**

Address of a two-longword array containing the starting and ending virtual address in the virtual address space of the process (either P0 or P1 regions) to be used as I/O buffers. The array contains, in order, the starting and ending virtual addresses. The address must specify an integral number of pages; that is, the low-order nine bits of each address must be 0. The pages must already exist and must be fully contained in either P0 or P1 space. All pages in the range must:

- Have identical page protection
- Be writable in the mode of the caller
- Be owned by the same access mode
- Be owned in a mode equal to or less privileged than the caller
- Be of the same page type (process or global)

DESCRIPTION

PTD$CREATE creates a new pseudoterminal with a unique device name. This device name is in the form FTA*n*:, where *n* is the unit number. This unit number is a VMS channel number that is used for control operations.

When a pseudoterminal is created, it inherits the current system terminal default attributes unless you specify an alternate set of characteristics.

RETURN VALUES

SS$_NORMAL	Normal successful completion.
SS$_ACCVIO	Unable to read one of the arguments.
SS$_BADPARAM	Bad parameter value.
SS$_EXBYTLM	Insufficient BYTLM to create device or map buffers.
SS$_EXQUOTA	Insufficient quota to create device.
SS$_EXASTLM	Insufficient AST quota for notification AST.
SS$_INSFMEM	Insufficient memory to create device.
SS$_INSFWSL	Insufficient working set limit to map buffers.
SS$_IVSECFLG	Invalid process or global section flags.
SS$_NOPRIV	No privilege for attempted operation.
SS$_PAGOWNVIO	Page owner violation.
SS$_VA_IN_USE	Virtual address already in use.

PTD$DELETE Delete a Pseudoterminal

Forces the pseudoterminal to be deleted and frees the channel.

FORMAT

PTD$CANCEL *chan*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

chan

VMS usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**
Number of the I/O channel assigned to the pseudoterminal.

DESCRIPTION

PTD$DELETE forces the pseudoterminal to be deleted and frees the channel assigned to the pseudoterminal. When a pseudoterminal is deleted, any process using the pseudoterminal (except the control program) is disconnected. PTD$DELETE request causes any pending I/O for the control program to be aborted. It deletes any queued event notification ASTs and returns the I/O buffers back to the application. It also causes the pseudoterminal unit control block (UCB) to be deleted once the reference count returns to zero.

RETURN VALUES

SS$_NORMAL	Normal successful completion.
SS$_DEVOFFLINE	Device is off line and request cannot proceed.
SS$_IVCHAN	Illegal channel.
SS$_NOPRIV	Insufficient privilege to perform request.

PTD$READ Read Data from Pseudoterminal

Reads data from the pseudoterminal.

FORMAT

PTD$READ *[efn], chan [,astadr] [,astprm] readbuf, readbuf_len*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

efn

VMS usage: **ef_number**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Number of the event flag to be set when PTD$READ returns the requested information. If you do not specify this argument, event flag 0 is used. When PTD$READ begins execution, it clears this flag.

chan

VMS usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Number of the I/O channel assigned to the pseudoterminal.

astadr

VMS usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed when PTD$READ completes. If you specify **astadr**, the AST routine executes at the same access mode as the caller of the PTD$READ routine.

astprm

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter to be passed to the AST service routine specified by the **astadr** argument.

readbuf

VMS usage: **char_string**
type: **character coded text string**
access: **write only**
mechanism: **by reference**

Address of the read I/O status longword. The first character position in an I/O buffer to receive all output is this address plus 4. The **readbuf** argument must be in the range specified in the **inadr** argument of the PTD$CREATE routine, otherwise an SS$_ACCVIO status is returned.

readbuf_len

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Number of characters that can be read from the pseudoterminal and stored in the buffer specified by **readbuf**. The low-order nine bits of the starting address plus **readbuf_len** must be less than or equal to 508. SS$_IVBUFLEN is returned if the value of **readbuf_len** is less than 0 or more than 508.

DESCRIPTION

The PTD$READ routine reads data from the pseudoterminal. The read request completes with a minimum of one character and a maximum of the number of characters specified by the **readbuf_len** argument. The read operation completes when the pseudoterminal has characters to output. If a read request is issued and no data is available, the read request is queued and then completed at a later time.

RETURN VALUES

SS$_NORMAL	Normal successful completion.
SS$_ACCVIO	Unable to read an argument, or invalid read buffer address.
SS$_DEVOFFLINE	Device is off line and request cannot proceed.
SS$_EXASTLM	Insufficient AST quota for notification AST.
SS$_ILLEFC	Illegal event flag cluster.
SS$_INSFMEM	Insufficient memory.
SS$_IVBUFLEN	Buffer size supplied is illegal.
SS$_IVCHAN	Illegal channel.
SS$_NOPRIV	Insufficient privilege to perform request.
SS$_UNASEFC	Unassociated event flag cluster.

PTD$SET_EVENT_NOTIFICATION Enable or Disable Terminal Event Notification ASTs

Enables or disables a number of repeating terminal event notification ASTs.

FORMAT

PTD$SET_EVENT_NOTIFICATION *chan, astadr [,astprm] [,acmode], type*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

chan

VMS usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**
Number of the I/O channel assigned to the pseudoterminal.

astadr

VMS usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**
Address of the notification AST service routine, or zero if the AST is to be canceled.

astprm

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
AST parameter to be passed to the AST service routine specified by the **astadr** argument.

acmode

VMS usage: **access_mode**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

Access mode for which the AST is to be declared. The most privileged access mode is the access mode of the caller. The resulting mode is the access mode at which the AST is declared.

type

VMS usage: **type_longword**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
Value that indicates which notification AST to enable. The $PTDDEF macro defines the symbolic names listed in Table C–2.

Table C–2 Symbolic Names Defined by $PTDDEF Macro

Symbolic Name	Description
PTD$C_SEND_XON	Deliver notification AST when pseudoterminal is ready to accept input. This AST is not delivered if the pseudoterminal is set to NO HOSTSYNC.
PTD$C_SEND_BELL	Deliver notification AST when pseudoterminal wants to stop input and signal it with a bell character.
PTD$C_SEND_XOFF	Deliver notification AST when pseudoterminal wants to stop input and signal it with a DC3 character.
PTD$C_STOP_OUTPUT	Deliver notification AST when pseudoterminal is stopping output.
PTD$C_RESUME_OUTPUT	Deliver notification AST when pseudoterminal is resuming output.
PTD$C_CHAR_CHANGED	Deliver notification AST when pseudoterminal has changed some device characteristic.
PTD$C_ABORT_OUTPUT	Deliver notification AST when pseudoterminal wants to abort output.
PTD$C_START_READ	Deliver notification AST when pseudoterminal is starting an application's read request. This AST is delivered only if read event notification has been enabled.
PTD$C_MIDDLE_READ	Deliver notification AST when pseudoterminal has finished sending an application's read request prompt string. This AST is delivered only if read event notification has been enabled.
PTD$C_END_READ	Deliver notification AST when pseudoterminal has finished an application's read request. This AST is delivered only if read event notification has been enabled.
PTD$C_ENABLE_READ	Enable terminal read event AST delivery. If this code is used, you cannot supply the **astadr** argument.
PTD$C_DISABLE_READ	Disable terminal read event AST delivery. If this code is used, you cannot supply the **astadr** argument.

DESCRIPTION

PTD$SET_EVENT_NOTIFICATION enables or disables the repeating terminal event notification ASTs listed in Table C–2. Once an event notification AST in enabled, it remains in effect until it is disabled or until the device is deleted.

RETURN VALUES

SS$_NORMAL	Normal successful completion.
SS$_ACCVIO	Unable to read an argument, or invalid I/O buffer address.
SS$_BADPARAM	An **astadr**, **astprm**, or **acmode** argument was not zero when enabling or disabling read notification.
SS$_DEVOFFLINE	Device is off line and request cannot proceed.
SS$_EXASTLM	Insufficient AST quota for notification AST.
SS$_INSFMEM	Insufficient memory.
SS$_IVCHAN	Illegal channel.
SS$_NOPRIV	Insufficient privilege to perform request.

PTD$WRITE Write Data to Pseudoterminal

Inputs data to the pseudoterminal and reads any immediately echoed characters.

FORMAT

PTD$WRITE *chan [,astadr] [,astprm], wrtbuf, wrtbuf_len [,echobuf] [,echobuf_len]*

RETURNS

VMS usage: **cond_value**
type: **longword (unsigned)**
access: **write only**
mechanism: **by value**

ARGUMENTS

chan

VMS usage: **channel**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**
Number of I/O channel assigned to the pseudoterminal.

astadr

VMS usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**
AST service routine to be executed when PTD$WRITE completes. If **astadr** is specified, the AST routine executes at the same access mode as the caller of the PTD$WRITE routine.

astprm

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
AST parameter to be passed to the AST service routine specified by the **astadr** argument.

wrtbuf

VMS usage: **char_string**
type: **character coded text string**
access: **read only**
mechanism: **by reference**
Address of the write I/O status longword. The first character in an I/O buffer to be written is this address plus 4. The **wrtbuf** must be in the range specified by the **inadr** argument of the PTD$CREATE routine; otherwise an SS$_ACCVIO status is returned.

wrtbuf_len

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Number of characters to be written to the pseudoterminal. These characters appear as input to the terminal side of the pseudoterminal. The low-order nine bits of the starting address plus **wrtbuf_len** must be less than or equal to 508. SS$_IVBUFLEN is returned if the value of **wrtbuf_len** is less than 0 or more than 508.

echobuf

VMS usage: **char_string**
type: **character coded text string**
access: **write only**
mechanism: **by reference**

Address of the echo I/O status longword. The first character position in an I/O buffer to receive all output is this address plus 4. The **echobuf** must be in the range specified by the **inadr** argument of the PTD$CREATE routine; otherwise an SS$_ACCVIO status is returned.

echobuf_len

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Number of characters that can be read from the pseudoterminal. If an echo buffer is specified, up to **echobuf_len** characters can be stored in it. The low-order nine bits of the starting address plus **echobuf_len** must be less than or equal to 508. SS$_IVBUFLEN is returned if 0 characters or more than 508 characters are specified.

DESCRIPTION

PTD$WRITE inputs data to the pseudoterminal and reads any immediately echoed characters. PTD$WRITE allows you to specify a buffer to receive any output generated by the write; you do not need to issue a separate read request to read this data.

RETURN VALUES

SS$_NORMAL	Normal successful completion.
SS$_ACCVIO	Unable to read an argument, or invalid I/O buffer address.
SS$_DATALOST	The terminal driver type-ahead buffer is full and character written was lost.
SS$_DATAOVERUN	The terminal driver type-ahead buffer is getting full; attempts to send more data might result in loss of characters.
SS$_DEVOFFLINE	Device is off line and request cannot proceed.
SS$_EXASTLM	Insufficient AST quota for notification AST.

SS$_INSFMEM	Insufficient memory.
SS$_IVBUFLEN	Buffer size supplied is illegal.
SS$_IVCHAN	Illegal channel.
SS$_NOPRIV	Insufficient privilege to perform request.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

O

P

Q

R

S

T

U

W

X

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	———	Local Digital subsidiary or approved distributor
Internal[1]	———	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VMS I/O User's Reference Manual: Part I
AA–LA84B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

Do Not Tear - Fold Here

Cut Along Dotted Line

Reader's Comments

VMS I/O User's Reference Manual: Part I
AA–LA84B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

Do Not Tear - Fold Here

Cut Along Dotted Line