OpenVMS Wide Area Network I/O User's Reference Manual

Order Number: AA–PWC7A–TE

May 1993

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

Revision/Update Information: This is a new manual.

Software Version: OpenVMS VAX Version 6.0

Digital Equipment Corporation
Maynard, Massachusetts

May 1993

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

ZK6108

This document was prepared using VAX DOCUMENT, Version 2.1.

Contents

5 Asynchronous DDCMP Interface Driver

A I/O Function Codes

Index

Examples

Figures

Tables

Preface

Intended Audience

This manual is intended for system programmers who want to save time and space by using I/O devices directly. If you do not require such detailed knowledge of the I/O drivers, use the device-independent services described in the *OpenVMS Record Management Services Reference Manual*. Readers are expected to have some experience with VAX MACRO or another high-level assembly language.

Document Structure

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

- Chapters 1 through 5 describe the use of communications device drivers supported by the OpenVMS operating system.
  - Chapter 1 discusses the DMC11/DMR11 interface driver.
  - Chapter 2 discusses the DMP11 and DMF32 interface drivers.
  - Chapter 3 discusses the DR11–W and DRV11–WA interface drivers.
  - Chapter 4 discusses the DR32 interface driver.
  - Chapter 5 discusses the asynchronous DDCMP interface driver.
- Appendix A summarizes the function codes, arguments, and function modifiers used by these drivers.

Associated Documents

For additional information, refer to the following documents:

- *OpenVMS I/O User's Reference Manual*
- *OpenVMS VAX Card Reader, Line Printer, and LPA11–K I/O User's Reference Manual*
- *OpenVMS System Services Reference Manual*
- *VAX FORTRAN User's Guide*
- *OpenVMS Programming Concepts Manual*
- *OpenVMS Record Management Services Reference Manual*
- *DECnet for OpenVMS Networking Manual*
- *VAX–11 2780/3780 Protocol Emulator User's Guide*
- OpenVMS system messages documentation
- *OpenVMS VAX Device Support Manual*

Conventions

In this manual, every use of OpenVMS VAX means the OpenVMS VAX operating system.

The following conventions are also 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 enclosed in a box indicates 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 optional elements. You can choose one, none, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS 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.
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 Bookreader versions of the manual.
italic text	Italic text emphasizes important information, indicates variables, and indicates complete titles of manuals. Italic text also represents information that can vary in system messages (for example, Internal error *number*), command lines (for example, /PRODUCER=*name*), and command parameters in text.

UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
-	A hyphen in code examples indicates that additional arguments to the request are provided on the line that follows.
numbers	All numbers in the text are assumed to be decimal, unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1

DMC11/DMR11 Interface Driver

This chapter describes the use of the DMC11 synchronous communications line interface driver in the OpenVMS VAX environment. (The DMR11 synchronous communications line interface uses the same driver in DMC compatibility mode; references to the DMC11 driver also imply the use of the DMR11 driver operating in DMC11 compatibility mode.) The DMC11 provides a direct-memory-access (DMA) interface between two computer systems using the Digital Data Communications Message Protocol (see Section 1.1.1). The DMC11 supports DMA data transfers of up to 16K bytes at rates of up to 1 million baud for local operation over coaxial cable and 56,000 baud for remote operation using modems. Both full- and half-duplex modes are supported.

The DMC11 is a message-oriented communications line interface used primarily to link two separate but cooperating computer systems.

1.1 Supported DMC11 Synchronous Line Interfaces

Table 1–1 lists the DMC11 options supported by the OpenVMS VAX operating system.

Table 1–1 Supported DMC11 Options

Type	Use
DMC11-AR with DMC11-FA DMC11-AR with DMC11-DA	Remote DMC11 and EIA or V35/DDS line unit
DMC11-AL with DMC11-MD DMC11-AL with DMC11-MA	Local DMC11 and 1M bps or 56 bps

1.1.1 Digital Data Communications Message Protocol (DDCMP)

To ensure reliable data transmission, the Digital Data Communications Message Protocol (DDCMP) has been implemented, using a high-speed microprocessor. For remote operations, a DMC11 can communicate with a different type of synchronous interface (or even a different type of computer), provided the remote system has implemented DDCMP.

DDCMP detects errors on the communication line connecting the systems using a 16-bit cyclic redundancy check (CRC). Errors are corrected, when necessary, by automatic message retransmission. Sequence numbers in message headers ensure that messages are delivered in the proper order with no omissions or duplications.

The DDCMP specification (Order No. AA–K175A–TC) provides more detailed information about DDCMP.

1.2 Driver Features and Capabilities

DMC11 driver capabilities include the following:

- A nonprivileged QIO interface to the DMC11 (allows use of the DMC11 as a raw-data channel)
- Unit attention conditions transmitted through attention ASTs and mailbox messages
- Both full- and half-duplex operation
- Interface design common to all communications devices supported by the OpenVMS VAX operating system
- Error logging of all DMC11 microprocessor and line unit errors
- Online diagnostics
- Separate transmit and receive quotas
- Assignment of several read buffers to the device

The following sections describe mailbox usage and I/O quotas.

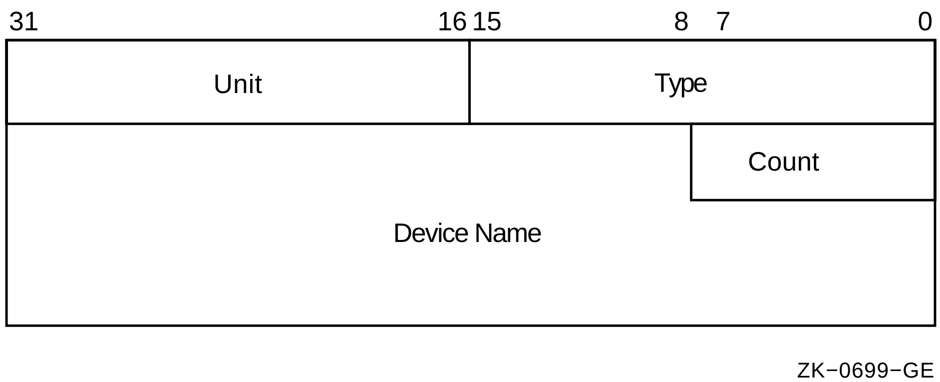
1.2.1 Mailbox Usage

The device owner process can associate a mailbox with a DMC11 by using the Assign I/O Channel ($ASSIGN) system service. (See the *OpenVMS System Services Reference Manual*.) The mailbox is used to receive messages that signal attention conditions about the unit. As illustrated in Figure 1–1, these messages have the following content and format:

- Message type. This can be any one of the following:
  - MSG$_XM_DATAVL—Data is available.
  - MSG$_XM_SHUTDN—The unit has been shut down.
  - MSG$_XM_ATTN—A disconnect, timeout, or data check occurred.

  The $MSGDEF macro is used to define message types.
- Physical unit number of the DMC11.
- Size (count) of the ASCII device name string.
- Device name string.

Figure 1–1 Mailbox Message Format



1.2.2 Quotas

Transmit operations are considered direct I/O operations and are limited by the process's direct I/O quota.

The quotas for the receive buffer free list (see Section 1.4.3.4) are the process's buffered I/O count and buffered I/O byte limit. After startup, the transient byte count and the buffered I/O byte limit are adjusted.

1.2.3 Power Failure

When a system power failure occurs, no DMC11 recovery is possible. The device is in a fatal error state and is shut down.

1.3 Device Information

You can obtain information about DMC11/DMR11 device characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *OpenVMS System Services Reference Manual*.)

$GETDVI returns DMC11/DMR11 device characteristics when you specify the item code DVI$_DEVCHAR. Table 1–2 lists these characteristics, which are defined by the $DEVDEF macro.

DVI$_DEVTYPE and DVI$_DEVCLASS return the device type and class names, which are defined by the $DCDEF macro. The device type for the DMC11 is DT$_DMC11; the device type for the DMR11 is DT$_ DMR11 (only after the device has been started once). The device class for the DMC11 is DC$_SCOM.

DVI$_DEVBUFSIZ returns the maximum message size. The maximum message size is the maximum send or receive message size for the unit. Messages greater than 512 bytes on modem-controlled lines are more prone to transmission errors and therefore may require more retransmissions.

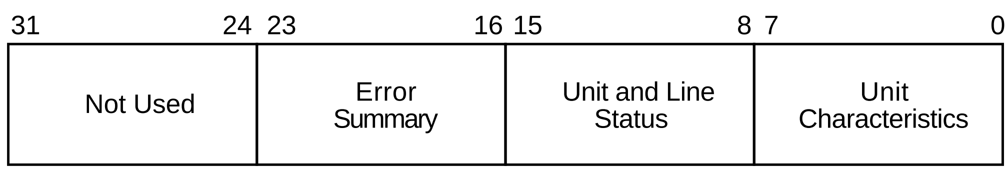
Table 1–2 DMC11/DMR11 Device Characteristics

Characteristic[1]	Meaning
Dynamic Bit (Conditionally Set)	
DEV$M_NET	Network device
Static Bits (Always Set)	
DEV$M_ODV	Output device
DEV$M_IDV	Input device

[1]Defined by the $DEVDEF macro

DVI$_DEVDEPEND returns the DMC11/DMR11 unit characteristics bits, the unit and line status bits, and the error summary bits in a longword field, as shown in Figure 1–2.

Figure 1–2 DVI$_DEVDEPEND Returns



ZK−5930−GE

The unit characteristics bits govern the DDCMP operating mode. They are defined by the $XMDEF macro and can be read or set. Table 1–3 lists the unit characteristics values and their meanings.

Table 1–3 DMC11/DMR11 Unit Characteristics

Characteristic	Meaning[1]
XM$M_CHR_MOP	DDCMP maintenance mode.
XM$M_CHR_SLAVE	DDCMP half-duplex slave station mode.
XM$M_CHR_HDPLX	DDCMP half-duplex mode.
XM$M_CHR_LOOPB	DDCMP loopback mode.
XM$M_CHR_MBX	The status of the mailbox associated with the unit. If this bit is set, the mailbox is enabled to receive messages signaling unsolicited data. (This bit can also be changed as a subfunction of read or write functions.)

[1]Section 1.1.1 describes DDCMP.

The status bits show the status of the unit and the line. The values are defined by the $XMDEF macro. They can be read, set, or cleared as indicated. Table 1–4 lists the status values and their meanings.

Table 1–4 DMC11/DMR11 Unit and Line Status

Status	Meaning
XM$M_STS_ACTIVE	Protocol is active. This bit is set when IO$_SETMODE!IO$_STARTUP is complete and is cleared when the unit is shut down (read only).
XM$M_STS_TIMO	Timeout. If set, indicates that the receiving computer is unresponsive (read or clear).
XM$M_STS_ORUN	Data overrun. If set, indicates that a message was received but lost because there is no receive buffer (read or clear).
XM$M_STS_DCHK	Data check. If set, indicates that a retransmission threshold has been exceeded (read or clear).
XM$M_STS_DISC	Disconnect. If set, indicates that the data set ready (DSR) modem line went from on to off (read or clear).

The error summary bits are set only when the driver must shut down the DMC11 interface because a fatal error occurred. These are read-only bits that are cleared by any of the IO$_SETMODE functions (see Section 1.4.3). The XM$M_STS_ACTIVE status bit is clear if any error summary bit is set. Table 1–5 lists the error summary bit values and their meanings.

Table 1–5 DMC11/DMR11 Error Summary Bits

Error Summary Bit	Meaning
XM$M_ERR_MAINT	DDCMP maintenance message was received.
XM$M_ERR_START	DDCMP START message was received.
XM$M_ERR_LOST	Data was lost when a message was received that was longer than the specified maximum message size.
XM$M_ERR_FATAL	An unexpected hardware or software error occurred.

1.4 DMC11 Function Codes

The basic DMC11 function codes are read, write, and set mode. All three functions take function modifiers.

1.4.1 Read

The operating system provides the following read function codes:

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

Received messages are multibuffered in system dynamic memory and then copied to the user’s address space when the read operation is performed.

The read functions take the following two device/function-dependent arguments:

- P1—The starting virtual address of the buffer that is to receive data
- P2—The size of the receive buffer in bytes

The read functions can take the following function modifiers:

- IO$M_DSABLMBX—Disables use of the associated mailbox for unsolicited data notification
- IO$M_NOW—Completes the read operation immediately if no message is available

1.4.2 Write

The operating system provides the following write function codes:

- IO$_WRITELBLK—Write logical block
- IO$_WRITEPBLK—Write physical block
- IO$_WRITEVBLK—Write virtual block

Transmitted messages are sent directly from the requesting process's buffer.

The write functions take the following device- or function-dependent arguments:

- P1—The starting virtual address of the buffer containing the data to be transmitted
- P2—The size of the buffer in bytes

The message size specified by P2 cannot be larger than the maximum send message size for the unit (see Section 1.3). If a message larger than the maximum size is sent, a status of SS$_DATAOVERUN is returned in the I/O status block.

The write functions can take the following function modifier:

- IO$M_ENABLMBX—Enable use of the associated mailbox

1.4.3 Set Mode

Set mode operations are used to perform protocol, operational, and program and driver interface operations with the DMC11. The operating system defines the following types of set mode functions:

- Set mode
- Set characteristics
- Enable attention AST
- Set mode and shut down unit
- Set mode and start unit

1.4.3.1 Set Mode and Set Characteristics

The set mode and set characteristics functions set device characteristics such as maximum message size. The operating system provides the following function codes:

- IO$_SETMODE—Set mode (no I/O privilege required)
- IO$_SETCHAR—Set characteristics (requires physical I/O privilege)

These two functions take the following device- or function-dependent argument:

- P1—The virtual address of the quadword characteristics buffer block if the characteristics are to be set. If this argument is zero, only the unit status and characteristics are returned in the I/O status block (see Section 1.5). Figure 1–3 shows the P1 characteristics block.

Figure 1–3 P1 Characteristics Block

31–24	23–16	15–8	7–0
Maximum Message Size		Type	Class
TPI	Error Summary	Status	Characteristics

ZK−0701−GE

In the buffer designated by P1 the device class is DC$_SCOM. Section 1.3 describes the device types. The maximum message size describes the maximum send or receive message size.

The second longword contains device- or function-dependent characteristics: unit characteristics, status, error summary bits, and transmit pipeline count (TPI). Any of the characteristics values and some of the status values can be set or cleared (see Tables 1–3, 1–4, and 1–5).

If the unit is active (XM$M_STS_ACTIVE is set), the action of a set mode or set characteristics function with a characteristics buffer is to clear the status bits or the error summary bits. If the unit is not active, the status bits or the error summary bits can be cleared, and the maximum message size, type, device class, unit characteristics, and transmit pipeline count can be changed.

1.4.3.2 Enable Attention AST

The enable attention AST function enables an AST to be queued when an attention condition occurs on the unit. An AST is queued when the driver sets or clears either an error summary bit or any of the unit status bits, or when a message is available and there is no waiting read request. The enable attention AST function is legal at any time, regardless of the condition of the unit status bits.

The operating system provides the following function codes:

- IO$_SETMODE!IO$M_ATTNAST—Enable attention AST
- IO$_SETCHAR!IO$M_ATTNAST—Enable attention AST

Enable attention AST enables an AST to be queued one time only. After the AST occurs, it must be explicitly reenabled by the function before the AST can occur again. The function code is also used to disable the AST. The function is subject to AST quotas.

The enable attention AST functions take the following device- or function-dependent arguments:

- P1—Address of AST service routine or 0 for disable
- P2—Ignored
- P3—Access mode to deliver AST

The AST service routine is called with an argument list. The first argument is the current value of the device- or function-dependent characteristics longword shown in Figure 1–3. The access mode specified by P3 is maximized with the requester’s access mode. (See the *OpenVMS System Services Reference Manual* for an explanation of this concept.)

1.4.3.3 Set Mode and Shut Down Unit

The set mode and shut down unit function stops the operation on an active unit (XM$M_STS_ACTIVE must be set) and then resets the unit characteristics.

The operating system provides the following function codes:

- IO$_SETMODE!IO$M_SHUTDOWN—Shut down unit
- IO$_SETCHAR!IO$M_SHUTDOWN—Shut down unit

These functions take the following device- or function-dependent argument:

- P1—The virtual address of the quadword characteristics block (Figure 1–3) if modes are to be set after shutdown. P1 is 0 if modes are not to be set after shutdown.

Both functions stop the DMC11 microprocessor and release all outstanding message blocks; any messages that have not been read are lost. The characteristics are reset after shutdown. Except for the sending of attention ASTs and mailbox messages, these functions act the same as the driver does when shutdown occurs because of a fatal error.

1.4.3.4 Set Mode and Start Unit

The set mode and start unit function sets the characteristics and starts the protocol on the associated unit. The operating system provides the following function codes:

- IO$_SETMODE!IO$M_STARTUP—Start unit
- IO$_SETCHAR!IO$M_STARTUP—Start unit

These functions take the following device- or function-dependent arguments:

- P1—The virtual address of the quadword characteristics block (Figure 1–3) if the characteristics are to be set. Characteristics are set before the device is started.
- P2—Ignored.
- P3—The number of preallocated receive-message blocks to ensure the availability of buffers to receive messages.

The total quota taken from the process's buffered I/O byte count quota is the DMC11 work space plus the number of receive-message buffers specified by P3 times the maximum message size. For example, if six 200-byte buffers are required, the total quota taken is 1456 bytes:

```
   256  (DMC11 work space)
+ 1200  (number of buffers X buffer size)
  ----
  1456  (total quota taken)
```

This quota is returned to the process when shutdown occurs.

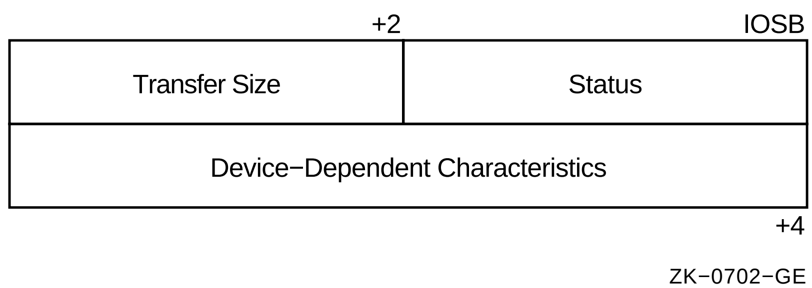
Receive-message blocks are used by the driver to receive messages that arrive independent of read-request timing. When a message arrives, it is matched with any outstanding read requests. If there are no outstanding read requests, the message is queued, and an attention AST or mailbox message is generated. (IO$_SETMODE!IO$M_ATTNAST or IO$_SETCHAR!IO$M_ATTNAST must be set to enable an attention AST; IO$M_ENABLMBX must be used to enable a mailbox message.)

When read, the receive-message block is returned to the receive-message **free list** defined by P3. If the free list is empty, no receive messages are possible. In this case, a data-lost condition can be generated if a message arrives. This nonfatal condition is reported by device-dependent data and an attention AST.

1.5 I/O Status Block

The I/O status block (IOSB) usage for all DMC11 functions is shown in Figure 1–4. Appendix A lists the status returns for these functions. (The OpenVMS system messages documentation provides explanations and suggested user actions for these returns.)

Figure 1–4 IOSB Contents for DMC11 Functions



In Figure 1–4, the transfer size at IOSB+2 is the actual number of bytes transferred. Table 1–3 lists the device-dependent characteristics returned at IOSB+4. These characteristics can also be obtained by using the Get Device/Volume Information ($GETDVI) system service (see Section 1.3).

1.6 Programming Example

The following sample program (Example 1–1) shows the typical use of QIO functions, such as transmitting and receiving data and checking for errors, in DMC11/DMR11 driver operations.

Example 1–1 DMC11/DMR11 Program Example

```
        .TITLE  EXAMPLE - DMC11/DMR11 Device Driver Sample Program
        .IDENT ’X00’

        $IODEF                  ; Define I/O functions and modes
        $XMDEF                  ; Define driver status flags
;
;  Macro definitions
;
        .macro  type    string,?L                 ;
        store   <string>                          ;
        movl    #$$.tmpx,cmdorab+rab$l_rbf        ;
        movw    #$$.tmpx1,cmdorab+rab$w_rsz       ;
        $put    rab=cmdorab                       ;
        blbs    r0,L                              ;
        $exit_s                                   ;
L:                                                ;
        .endm   type                              ;
```

(continued on next page)

Example 1–1 (Cont.) DMC11/DMR11 Program Example

```
;
        .macro  store   string,pre
        .save
        .psect  $$$DEV
        $$.tmpx=.
        pre
        .ascii  %string%
        $$.tmpx1=.-$$.tmpx
        .restore
        .endm   store

CMDOFAB:        $FAB    fac=put,fnm=sys$output:,-  ; Output FAB
                mrs=132,rat=cr,rfm=var
CMDORAB:        $RAB    ubf=cmdbuf,usz=cmdbsz,-    ; Output RAB
                fab=cmdofab
CMDBUF::        .BLKB   256                     ; Command buffer
CMDBSZ=         .-CMDBUF                        ; Buffer size
FAOBUFDSC:      .LONG   CMDBSZ,CMDBUF           ; FAO buffer
                                                ; descriptor
FAOLEN:         .BLKL   1                       ; FAO output buffer
                                                ; length
P2BUF::         .BLKL   50                      ; P2 buffer
P2BUFSZ=        .-P2BUF                         ; P2 buffer size
P2BUFDSC:       .LONG   P2BUFSZ,P2BUF           ; P2 buffer descriptor
P1BUF::         .BLKQ   1                       ; P1 buffer
P1BUFSZ=        .-P1BUF                         ; P1 buffer size
CHNL::          .BLKL   1                       ; Channel number
IOSB::          .BLKQ   1                       ; I/O status block
DEVDSC:         .ASCID  'DEV'                   ; Device to assign
QIOREQDSC:      .LONG   QIOREQSZ,QIOREQ         ; QIO request status
QIOREQ:         .ASCII  'QIO completion status = !XL'
                .ASCII  'IOSB1 = !XL, IOSB2 = !XL'
QIOREQSZ=       .-QIOREQ                        ; Size of QIO status
                                                ; report
XMTBUFLEN=512                                   ; Size of transmit
                                                ; buffer
XMTBUF:         .REPEAT XMTBUFLEN
                .BYTE   ^X93                    ; Transmit data
                .ENDR
RCVBUF:         .BLKB   XMTBUFLEN

;
; This is the start of the program section.
;
START:: .WORD   0
        $OPEN   FAB=CMDOFAB                     ; Open output
        BLBC    R0,EXIT                         ;
        $CONNECT RAB=CMDORAB                    ; Connect to output
        BLBC    R0,EXIT                         ;
        BRB     CONT                            ; Continue
EXIT:   $EXIT_S                                 ; Exit program
```

(continued on next page)

Example 1–1 (Cont.) DMC11/DMR11 Program Example

```
CONT:   TYPE    <DMC11/DMR11 Test Program>
        TYPE    <>
        $ASSIGN_S       DEVNAM=DEVDSC,CHAN=CHNL ; Assign unit
        BLBC    R0,EXIT                         ; Exit on error
;
; Initialize and start controller
;
        MOVZBL  #XM$M_CHR_LOOPB,P1BUF+4         ; Set P1 flags -
                                                ; Loopback
        MOVW    #XMTBUFLEN,P1BUF+2              ; Set P1 buffer size
        CLRL    P2BUFDSC                        ; Set zero length P2
                                                ; buffer
        BSBW    INIT                            ; Issue QIO
;
; Loopback data
;
        MOVZWL  #100,R9                         ; Loop device 100
                                                ; times
10$:    BSBW    XMIT                            ; Issue transmit
        BSBW    RECV                            ; Issue receive
        MOVAB   XMTBUF,R1                       ; Get address of xmit
                                                ; data
        MOVAB   RCVBUF,R2                       ; Get address of
                                                ; received data
        MOVZWL  #XMTBUFLEN,R3                   ; Get number of bytes
                                                ; to verify
20$:    CMPB    (R1)+,(R2)+                     ; Check data
        BNEQ    30$                             ;
        SOBGTR  R3,20$                          ;
        SOBGTR  R9,10$                          ;
        BRW     EXIT                            ; Exit

30$:    TYPE    <*** Loopback buffer comparison error ***>
        BRW     EXIT                            ; Exit
;
; Initialize controller QIO
;
INIT:   TYPE    <*** Initialize controller QIO ***>     ;
        $QIOW_S chan=chnl,func=#io$_setchar!io$m_startup,-
                p1=p1buf,p2=#p2bufdsc,iosb=iosb,p3=#5   ;
        BRW     QIO_STATUS                      ;
```

(continued on next page)

Example 1–1 (Cont.) DMC11/DMR11 Program Example

```
;
; Xmit data QIO
;
XMIT:   TYPE    <*** Transmit buffer QIO ***>   ;
        $QIO_S  chan=chnl,func=#io$writevblk,p1=xmtbuf,-
                p2=#xmtbuflen,iosb=iosb
        BRW     QIO_XMTST                       ;
;
; Receive data QIO
;
RECV:   TYPE    <*** Receive buffer QIO ***>    ;
        $QIOW_S chan=chnl,efn=#2,func=#io$_readvblk,-
                p1=rcvbuf,p2=#xmtbuflen,iosb=iosb
        .BRB    qio_status
        .ENABL  LSB
QIO_STATUS:                                     ; Check status of QIO
        BLBC    IOSB,10$                        ; Br if error on QIO
QIO_XMTST:                                      ; Check status of XMIT
        BLBC    R0,10$                          ; Br if error on
                                                ; request
        RSB                                     ; Else, return to
                                                ; caller

10$:    MOVZWL  IOSB,R1                         ; Get I/O status block
        PUSHL   R1                              ; Push I/O status block
        PUSHL   R0                              ; Push system service
                                                ; status
        PUSHAQ  FAOBUFDSC                       ; Push address of FAO
                                                ; buffer descriptor
        PUSHAW  FAOLEN                          ; Push address of
                                                ; output length
        PUSHAQ  QIOREQDSC                       ; Push address of
                                                ; input string
        CALLS   #5,@#SYS$FAO                    ; Get error message
        MOVAB   CMDBUF,CMDORAB+RAB$L_RBF        ; Get output buffer
                                                ; address
        MOVW    FAOLEN,CMDORAB+RAB$W_RSZ        ; Get output buffer
                                                ; length
        $PUT    CMDORAB                         ; Print error text
        BRW     EXIT                            ; Exit
        .DSABL  LSB

        .END    START
```

2 DMP11 and DMF32 Interface Drivers

This chapter describes the use of the DMP11 multipoint communications line interface and DMF32 synchronous line interface drivers in an OpenVMS VAX environment.

2.1 Supported Devices

The DMP11 multipoint communications line interface is a direct-memory-access (DMA) device that uses the Digital Data Communications Message Protocol (DDCMP) to provide direct communication between a VAX processor and DDCMP-compatible devices, such as other DMP11s and some terminals (for example, the VT62). Up to 32 devices can be connected to the DMP11 through a single, multidrop, DDCMP-compatible line.

The logical connection between the DMP11 and a connected device is called a **tributary**. In multipoint configurations, the DMP11 functions as a multipoint control station, and the devices on the DDCMP line are located at tributary addresses. A controller operating in tributary mode on this line is called a **tributary station**.

In point-to-point configurations, one DMP11 is connected to one other controller. Controllers in this mode are called **point-to-point stations**.

The DMF32 synchronous line interface is a DMA communications device that uses a software implementation of DDCMP to provide an interface between a VAX processor and other DDCMP-compatible devices, such as a DMP11 or DMC11. The DMF32 supports both full- and half-duplex modes as well as tributary mode on a multidrop DDCMP-compatible line.

In a multipoint configuration, the DMF32 operates in tributary mode and is located at a tributary address on the DDCMP line.

In point-to-point configurations, one DMF32 is connected to one other controller. Controllers in this mode are called point-to-point stations.

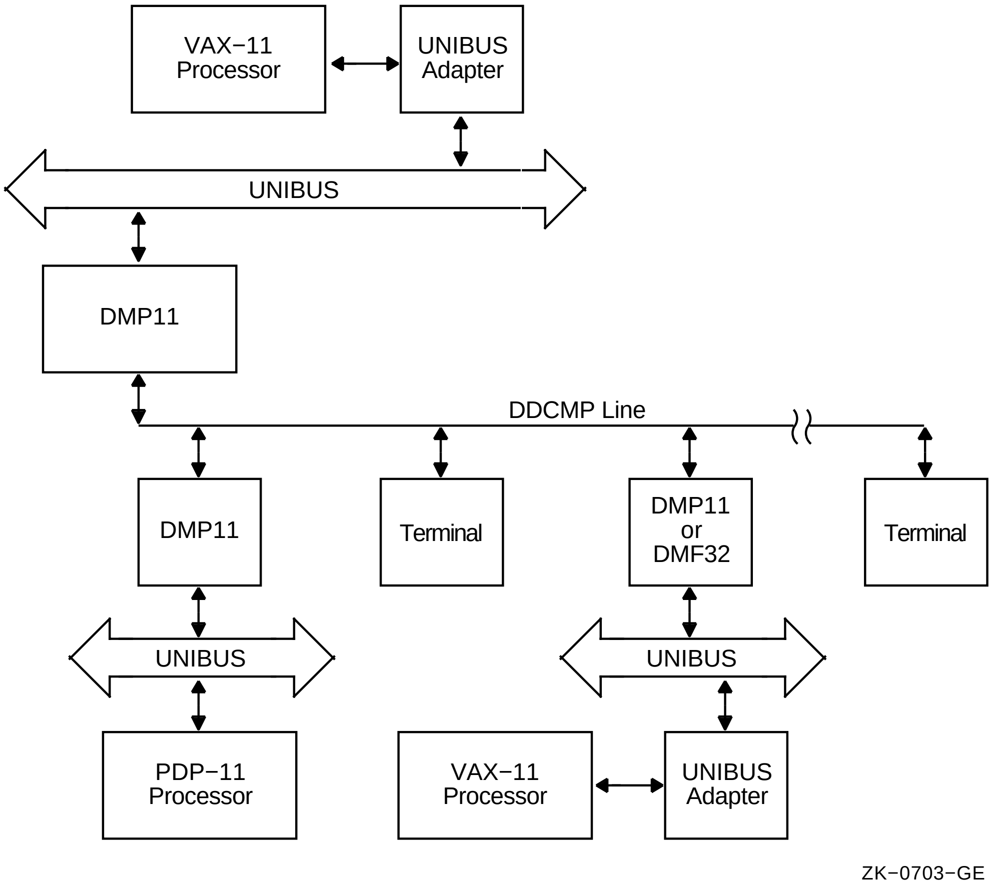
Figure 2–1 shows a typical DMP11/DMF32 multipoint configuration.

2.2 Driver Features and Capabilities

The DMP11 and DMF32 drivers provide the following capabilities:

- Multipoint operating mode in which the DMP11 functions as a control station connected to 1 to 32 devices and tributary stations (not for the DMF32 driver)
- Multipoint operating mode in which the DMP11 or DMF32 functions as a tributary station

Figure 2–1 Typical DMP11/DMF32 Multipoint Configuration



- Point-to-point operating mode in which the DMP11 or DMF32 is connected to one other controller also operating in point-to-point mode
- DMC11-compatible operating mode in which the DMP11 is connected to either a DMC11, a DMR11, another synchronous line interface using DDCMP, or another DMP11 running in DMC11-compatible mode (not for the DMF32 driver)
- Support for using the DMF32 in high-level data link control (HDLC) bit stuff mode
- Support for using a general character-oriented protocol over the DMF32
- A nonprivileged QIO interface to the DMP11 and DMF32 for using these devices as raw-data channels
- Tributary attention conditions transmitted through attention ASTs
- Full- and half-duplex operation
- Interface design common to all communications devices supported by the OpenVMS VAX operating system
- Separate transmit and receive queues
- Assignment of multiple read and write buffers to the device

2.2.1 Character-Oriented Protocols and HDLC Bit Stuff Mode

DMF32 synchronous line unit supports character-oriented protocols and the high-level data link control (HDLC) bit stuff mode. The DMF32 driver can transmit and receive a framed message and also provide some modem control. General protocol handling for the character-oriented protocols is supported at the DMF32 driver level. However, the DMF32 driver provides an interface to the higher level protocol so that receive messages are framed by the rules of the protocol. For HDLC mode, you can transmit and receive frame messages in full-duplex mode only.

Sections 2.4.3.2 through 2.4.3.5 describe these features of the DMF32 driver in greater detail.

2.2.2 Quotas

Transmit operations are direct (DMP11) or buffered (DMF32) I/O operations and are limited by the process's direct or buffered I/O quota.

The quotas for the receive buffer free list (see Section 2.4.3.1) are the process's buffered I/O quota and buffered I/O byte count quota.

2.2.3 Power Failure

If a system power failure occurs, no DMP11 or DMF32 recovery is possible. The driver is in a fatal error state and shuts down.

2.3 Device Information

You can obtain information about DMP11 or DMF32 characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *OpenVMS System Services Reference Manual*.) $GETDVI returns device characteristics when you specify the item code DVI$_DEVCHAR. Table 2–1 lists these characteristics, which are defined by the $DEVDEF macro.

Table 2–1 DMP11 and DMF32 Device Characteristics

Characteristic[1]	Meaning
	Static Bits (Always Set)
DEV$M_NET	Network device. Set for terminal port if it is a network device.
DEV$M_AVL	Available device. Set when unit control block (UCB) is initialized.
DEV$M_ODV	Output device.
DEV$M_IDV	Input device.
DEV$M_SHR[2]	Shareable device.

[1]Defined by the $DEVDEF macro
[2]Only for DMP11

DVI$_DEVCLASS returns the device class, which is DC$_SCOM. DVI$_DEFTYPE returns the device type, which is DT$_DMP11 for the DMP11 and DT$_DMF32 for the DMF32. The $DCDEF macro defines the device class and device type names.

DVI$_DEVBUFSIZ returns the maximum message size. The maximum message size is the maximum send or receive message size for the unit. Messages greater than 512 bytes on modem-controlled lines are more prone to transmission errors.

DVI$_DEVDEPEND returns the unit characteristics bits, the unit and line status bits, the error summary bits, and the specific errors in a longword field, as shown in Figure 2–2.

Figure 2–2 DVI$_DEVDEPEND Returns

31 24	23 16	15 8	7 0
Error	Error Summary	Unit and Line Status	Unit Characteristics

ZK−5931−GE

Unit characteristics bits govern the DDCMP operating mode. They are defined by the $XMDEF macro and can be set by a set mode function (see Section 2.4.3.1) or can be read by a sense mode function (see Section 2.4.4). Table 2–2 lists the unit characteristics values and their meanings.

Table 2–2 DMP11 and DMF32 Unit Characteristics

Characteristic	Meaning
XM$M_CHR_MOP	Specifies DDCMP maintenance mode
XM$M_CHR_LOOPB	Specifies loopback mode
XM$M_CHR_HDPLX	Specifies half-duplex operation
XM$M_CHR_CTRL[1]	Specifies control station
XM$M_CHR_TRIB	Specifies tributary station
XM$M_CHR_DMC[1]	Specifies DMC11-compatible mode

[1]Only for DMP11

The status bits show the status of the unit and the line. These bits can be set or cleared only when the controller and tributary are not active.

Table 2–3 lists the status values and their meanings. The values are defined by the $XMDEF macro.

Table 2–3 DMP11 and DMF32 Unit and Line Status

Status	Meaning
XM$M_STS_ACTIVE	DDCMP protocol is active.
XM$M_STS_DISC	Modem line went from on to off. This bit will be returned in the field IRP$L_IOST2 if the driver has had a timeout while waiting for the CTS signal to be present on the device.

(continued on next page)

Table 2–3 (Cont.) DMP11 and DMF32 Unit and Line Status

Status	Meaning
XM$M_STS_RUNNING[1]	Tributary is responding.
XM$M_STS_BUFFAIL	Receive buffer allocation failed.

[1]Only for DMP11

The error summary bits are set when an error occurs. If the error is fatal, the DMP11 or DMF32 is shut down. Table 2–4 lists the error summary bit values and their meanings.

Table 2–4 Error Summary Bits

Error Summary Bit[1]	Meaning
XM$M_ERR_MAINT	DDCMP maintenance message received
XM$M_ERR_START	DDCMP start message received
XM$M_ERR_FATAL	Hardware or software error occurred on controller
XM$M_ERR_TRIB	Hardware or software error occurred on tributary
XM$M_ERR_LOST	Data lost when a received message was longer than the specified maximum message size
XM$M_ERR_THRESH	Receive, transmit, or select threshold errors

[1]Read-only

Table 2–5 lists the errors that can be specified. These errors are mapped to the indicated codes.

Table 2–5 DMP11 and DMF32 Errors

Value[1] (octal)	Meaning	Code Set
2	Receive threshold error	XM$M_ERR_THRESH
4	Transmit threshold error	XM$M_ERR_THRESH
6	Select threshold error	XM$M_ERR_THRESH
10	Start received in run state	XM$M_ERR_START
12	Maintenance received in run state	XM$M_ERR_MAINT
14	Maintenance received in halt state	(none)
16	Start received in maintenance state	XM$M_ERR_START
22	Dead tributary	XM$M_STS_RUNNING[2] (cleared)
24	Running tributary	XM$M_STS_RUNNING[2] (set)
26	Babbling tributary	XM$M_ERR_TRIB

[1]Not provided on the DMF32
[2]Not supported for the DMF32

(continued on next page)

Table 2–5 (Cont.) DMP11 and DMF32 Errors

Value[1] (octal)	Meaning	Code Set
30	Streaming tributary	XM$M_ERR_TRIB
32	Ring detection	(none)
100–276	Internal procedure (software) errors	XM$M_ERR_TRIB
300	Buffer too small	XM$M_ERR_LOST
302	Nonexistent memory	XM$M_ERR_FATAL
304	Modem disconnected	XM$M_STS_DISC and XM$M_ERR_FATAL
306	Queue overrun	XM$M_ERR_FATAL[2]
310	Carrier lost on modem	XM$M_ERR_FATAL

[1]Not provided on the DMF32
[2]Not supported for the DMF32

2.4 DMP11 and DMF32 Function Codes

The DMP11 and DMF32 drivers can perform logical, virtual, and physical I/O operations. The basic functions are read, write, set mode, set characteristics, and sense mode. Table 2–6 lists these functions and their function codes. The sections that follow describe these functions in greater detail.

Table 2–6 DMP11 and DMF32 I/O Functions

Function Code	Arguments	Type[1]	Modifiers	Function
IO$_READLBLK	P1,P2	L	IO$M_NOW	Read logical block.
IO$_READVBLK	P1,P2	V	IO$M_NOW	Read virtual block.
IO$_READPBLK	P1,P2,[P6]	P	IO$M_NOW	Read physical block.
IO$_WRITELBLK	P1,P2	L		Write logical block.
IO$_WRITEVBLK	P1,P2	V		Write virtual block.
IO$_WRITEPBLK	P1,P2,[P6]	P		Write physical block.
IO$_CLEAN		L		Complete outstanding requests (character-oriented protocols), and abort outstanding transmits (bit stuff mode).
IO$_SETMODE	P1,[P2],P3	L	IO$M_CTRL IO$M_SHUTDOWN IO$M_STARTUP IO$M_ATTNAST IO$M_SET_MODEM[2]	Set DMP11 and DMF32 characteristics and controller state for subsequent operations.

[1]V = virtual, L = logical, P = physical (there is no functional difference in these operations)
[2]Only for DMP11

(continued on next page)

Table 2–6 (Cont.) DMP11 and DMF32 I/O Functions

Function Code	Arguments	Type[1]	Modifiers	Function
IO$_SETCHAR	P1,[P2],P3,[P6]	P	IO$M_CTRL IO$M_SHUTDOWN IO$M_STARTUP IO$M_ATTNAST IO$M_SET_MODEM[2]	Set DMP11 and DMF32 characteristics and controller state for subsequent operations.
IO$_SENSEMODE	P1,P2	L	IO$M_CTRL IO$M_RD_MEM[2] IO$M_RD_MODEM IO$M_RD_COUNTS IO$M_CLR_COUNTS	Sense controller or tributary characteristics and return them in specified buffers.

[1]V = virtual, L = logical, P = physical (there is no functional difference in these operations)

[2]Only for DMP11

Although the DMP11 and DMF32 drivers do not differentiate among logical, virtual, and physical I/O functions (all are treated identically), you must have the required privilege to issue a request.

2.4.1 Read

Read functions provide for the direct transfer of data into the user process's virtual memory address space. The operating system provides the following function codes:

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

Received messages are multibuffered in system dynamic memory and then copied to the user's buffer.

The read functions take the following device- or function-dependent arguments:

- P1—The starting virtual address of the buffer that is to receive data.
- P2—The size of the receive buffer in bytes.
- P6—The address of a diagnostic buffer; only for physical I/O functions (optional). See Section 2.4.5.

The message size specified by P2 cannot be larger than the maximum receive-message size for the unit (see Section 2.3). If a message larger than the maximum size is received, a status of SS$_DATAOVERUN is returned in the I/O status block.

The read functions can take the following function modifier:

- IO$M_NOW—Complete the read operation immediately with a received message. (If no message is currently available, return a status of SS$_ ENDOFFILE in the I/O status block.)

2.4.2 Write

Write functions provide for the direct transfer of data from the user process's virtual memory address space. The operating system provides the following function codes:

- IO$_WRITELBLK—Write logical block
- IO$_WRITEVBLK—Write virtual block
- IO$_WRITEPBLK—Write physical block

Transmitted DMP11 messages are sent directly from the requesting process's buffer. DMF32 messages are copied into a system buffer before they are transmitted.

The write functions take the following device- or function-dependent arguments:

- P1—The starting virtual address of the buffer containing the data to be transmitted.
- P2—The size of the buffer in bytes.
- P6—The address of a diagnostic buffer; only for physical I/O functions (optional). See Section 2.4.5.

The message size specified by P2 cannot be larger than the maximum send-message size for the unit (see Section 2.3).

The write functions take no function modifiers.

2.4.3 Set Mode and Set Characteristics

Set mode operations are used to perform protocol, operational, and program/driver interface operations with the DMP11 or DMF32 drivers. The operating system defines the following types of set mode functions:

- Set mode
- Set characteristics
- Set controller mode
- Set tributary mode
- Enable attention AST
- Shutdown controller
- Shutdown tributary

Used without function modifiers, set mode and set characteristics functions can modify an existing tributary. Used with certain function modifiers, they can perform DMP11 or DMF32 operations such as starting a tributary and requesting an attention AST. The operating system provides the following function codes:

- IO$_SETMODE—Set mode (no I/O privilege required)
- IO$_SETCHAR—Set characteristics (requires physical I/O privilege)

The other five types of set mode functions, which use the two function codes with certain function modifiers, are described in the sections that follow.

To use the IO$_SETMODE and IO$_SETCHAR functions, you must assign the appropriate unit control block (UCB) with the Assign I/O Channel ($ASSIGN) system service.

2.4.3.1 Set Controller Mode

The set controller mode function sets the DMP11 or DMF32 controller state and activates the controller. The following combinations of function code and modifier are provided:

- IO$_SETMODE!IO$M_CTRL—Set controller characteristics
- IO$_SETCHAR!IO$M_CTRL—Set controller characteristics
- IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP—Set controller characteristics and start the controller
- IO$_SETCHAR!IO$M_CTRL!IO$M_STARTUP—Set controller characteristics and start the controller

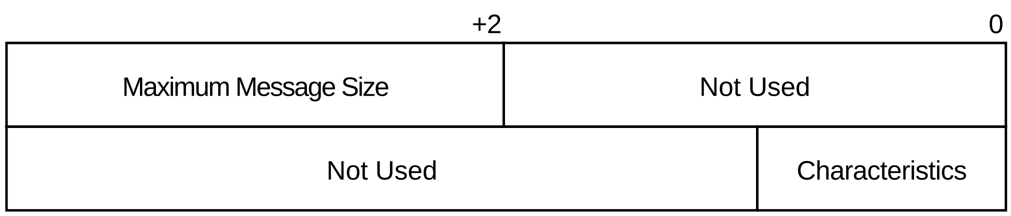
If the function modifier IO$M_STARTUP is specified, the controller is started and the modem is enabled. If IO$M_STARTUP is not specified, the specified characteristics are simply modified.

These codes take the following device- or function-dependent arguments:

- P1—The virtual address of a quadword characteristics buffer.
- P2—The address of a descriptor for an extended characteristics buffer (optional).
- P3—The number of preallocated receive-message blocks to allocate (referred to as the size of the **common receive pool**). See the NMA$C_PCLI_BFN parameter ID described in Table 2–8.

Figure 2–3 shows the format of the P1 characteristics buffer. The maximum message size in the first longword specifies the maximum allowable transmit and receive-message length.

Figure 2–3 P1 Characteristics Buffer (Set Controller)



ZK−0705−GE

Table 2–7 lists the DMP11 and DMF32 characteristics that can be set in the second longword. The $XMDEF macro defines these values.

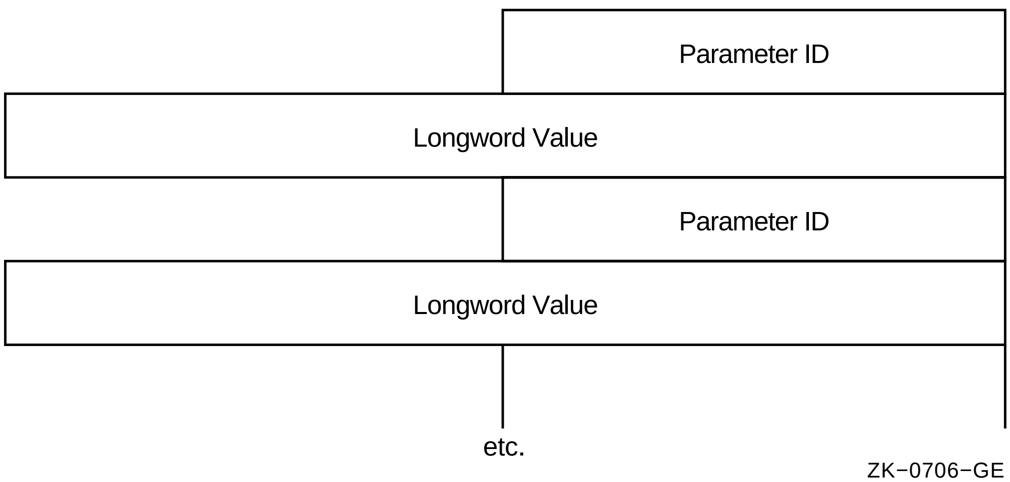
Table 2–7 DMP11 and DMF32 Characteristics

Characteristic	Meaning
XM$M_CHR_LOOPB	Sets loopback mode
XM$M_CHR_HDPLX	Sets half-duplex operation
XM$M_CHR_CTRL[1]	Specifies control station
XM$M_CHR_TRIB	Specifies tributary station
XM$M_CHR_DMC[1]	Specifies DMC11-compatible mode

[1]Only for DMP11

The P2 buffer consists of a series of six-byte entries. The first word contains the parameter identifier (ID), and the longword that follows it contains one of the values that can be associated with the parameter ID. Figure 2–4 shows the format for this buffer.

Figure 2–4 P2 Extended Characteristics Buffer (Set Controller)



If both P1 and P2 characteristics are specified, the P2 characteristics supersede the P1 characteristics. For example, if P1 specifies XM$M_CHR_CTRL and P2 specifies NMA$C_PCLI_PRO with a value of NMA$C_LINPR_TRIB (that is, a tributary), the device is started as a tributary.

Table 2–8 lists the parameter IDs and values that can be specified in the P2 buffer. The $NMADEF macro defines these values.

Section 2.4.3.2 lists the parameter IDs allowed for the character-oriented and HDLC bit stuff modes of operation.

Table 2–8 P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCLI_PRO	Protocol mode. The following values can be specified: **Value** / **Meaning** NMA$C_LINPR_POI — DDCMP point-to-point (default) NMA$C_LINPR_CON[1] — DDCMP control station NMA$C_LINPR_TRI — DDCMP tributary NMA$C_LINPR_DMC[1] — DDCMP DMC mode NMA$C_LINPR_LAPB[2] — HDLC bit stuff mode NMA$C_LINPR_BSY[2] — General character-oriented protocol mode
NMA$C_PCLI_DUP	Duplex mode. The following values can be specified: **Value** / **Meaning** NMA$C_DPX_FUL — Full-duplex (default) NMA$C_DPX_HAL — Half-duplex
NMA$C_PCLI_CON	Controller mode. The following values can be specified: **Value** / **Meaning** NMA$C_LINCN_NOR — Normal (default) NMA$C_LINCN_LOO — Loopback
NMA$C_PCLI_BFN	Number of receive buffers to preallocate. Must be provided here or as P3 argument.
NMA$C_PCLI_BUS	Maximum allowable transmit and receive message length (default = 512 bytes).
NMA$C_PCLI_NMS	Number of sync characters to precede message.
NMA$C_PCLI_SLT[1,3]	Number of milliseconds (msec) in the period of incrementing tributary priorities and the transmit delay (min = 50; default = 50).
NMA$C_PCLI_DDT[1,3]	Number of msec in the period of polling dead tributaries (default = 10000).
NMA$C_PCLI_DLT[1,3]	Number of msec between polls (default = 0).
NMA$C_PCLI_SRT[1,3]	Timer value used by control station and half-duplex point-to-point to establish that a tributary is streaming (default = 6000).

[1]Only for DMP11.

[2]Only for DMF32.

[3]A global polling parameter. All timer values must be specified in milliseconds.

2.4.3.2 Additional Features of the DMF32 Driver

The character-oriented protocols and the HDLC bit stuff mode do not have the concept of line and circuit. Therefore, only $QIO requests that include the function modifier IO$M_CTRL are allowed. The operating system does not acknowledge characteristics set in the P1 buffer for character-oriented and HDLC bit stuff modes of operation. You must have CMKRNL privilege to run the DMF32 in character-oriented mode. Only the parameters listed in Table 2–9 are relevant to the character-oriented and HDLC bit stuff modes of operation.

Table 2–9 P2 Extended Characteristics Values (DMF32 Driver)

Parameter ID	Meaning
NMA$C_PCLI_PRO	Must be set to NMA$C_LINPR_BSY to specify character-oriented mode of operation or to NMA$C_LINPR_LAPB to specify HDLC bit stuff mode.
NMA$C_PCLI_DUP	Requests full- or half-duplex mode of operation. (HDLC bit stuff mode supports full-duplex mode only.) If half-duplex mode is specified, the DMF32 driver sets the request to send (RTS) signal, waits for the clear to send (CTS) signal at the beginning of the transmit, and then drops RTS at the end of the transmit. The full-duplex mode value is NMA$C_DPX_FUL; the half-duplex mode value is NMA$C_DPX_HAL.
NMA$C_PCLI_BFN	The number of buffers the device can allocate for use as receive buffers. This value must be greater than 1. Default is 4.
NMA$C_PCLI_BUS	The size of the buffers to be allocated.
NMA$C_PCLI_CON	The state the controller is set to. If NMA$C_LINCN_NOR is specified, the device operates normally. If NMA$C_LINCN_LOO is specified, the device operates in internal loopback mode. Default is normal operation.
NMA$C_PCLI_SYC[1]	The sync character used by device. Defaults to 32 hexadecimal.
NMA$C_PCLI_NMS[1]	The number of sync characters to precede a transmit. Defaults to 8.
NMA$C_PCLI_BPC[1]	The number of bits per character (5, 6, 7, or 8). Defaults to 8.
NMA$C_PCLI_FRA[1]	The address of the protocol framing routine (in nonpaged pool). This parameter must be specified.
NMA$C_PCLI_STI1[1] NMA$C_PCLI_STI2[1]	These two parameters contain the initial value for the quadword of framing routine state information.
NMA$C_PCLI_MCL[1]	Determines whether modem signals should be turned off when a DEASSIGN operation is performed. The DMF32 driver always clears the modem signals on the last DEASSIGN. However, on all other DEASSIGN operations, the modem signals are cleared only if the value of NMA$C_PCLI_MCL is 0. If the value NMA$C_STATE_ON is specified, the data terminal ready (DTR) signal is dropped when DEASSIGN is performed. If the value NMA$C_STATE_OFF is specified, DTR is not dropped until the last DEASSIGN.
NMA$C_PCLI_TMO[1]	Specifies the timeout (in seconds) when waiting for CTS during transmit operations.

[1]Character-oriented mode only

2.4.3.3 Framing Routine Interface for Character-Oriented Protocols

In general, the character-oriented protocols each have their own rule for framing receive messages. To provide support for each protocol's special framing rules, the DMF32 driver has been extended to provide support for calling a special framing routine from the DMF32 driver's processing of receive messages. This routine is defined by the higher level software using the DMF32 driver and is loaded by that same software into nonpaged pool. The address of this routine is passed to the driver when the device is started up. The purpose of the framing routine is to tell the driver how to frame each byte of the received data message and to tell the driver that the received message is complete and ready to be posted.

The address of the framing routine is kept in the DMF32 driver's internal buffer. The internal buffer also contains a quadword that is used by the framing routine for holding state information while it is framing the receive message. The framing routine is called by the driver at FORK IPL through a JSB instruction. The input and the output to the framing routine is described in the following tables.

Input	Contents
R0	Address of quadword of state information.
R1 bits 0–7	Character to examine. The high-order bit is set if this is the first character of a new frame.

Output	Contents
R0	Status information for the DMF32 driver. The following bits are defined:

Value	Meaning
XG$V_BUFFER_CHAR	If clear, buffer the character in the next position. If set, use bit XG$V_BUFFER_IN_PREV_POS.
XG$V_BUFFER_IN_PREV_POS	If clear, ignore the character. If set, buffer the character in the previous position; do not update the buffer pointer.
XG$V_COMPLETE_READ	If clear, ignore. If set, return the framed buffer to user (buffer character if required).

After the DMF32 driver has completed a framed receive-data message, the driver resets the quadword of state information to the value passed when the device is started up. This means that the driver resets error information along with success information.

2.4.3.4 Using the DMF32 Driver Transmitter Interface in Character-Oriented Mode

For write requests made through the QIO interface, the P4 parameter contains the address of a quadword buffer to be used to update the field in the DMF32 driver's internal buffer, which contains the state information for the framing routine. If this parameter is 0, the state information is not updated.

If the DMF32 driver has had a timeout error while waiting for the CTS signal to be present on the device, the bit XM$M_STS_DISC is returned in the field IRP$L_IOST2.

2.4.3.5 IO$_CLEAN Function

The clean function either completes or aborts outstanding device requests. The operating system provides the following function code:

- IO$_CLEAN

For character-oriented protocols, a clean function request results in the completion of all outstanding I/O requests pending on the device. For HDLC bit stuff mode, a clean function request results in the aborting of all outstanding transmit operations on the device. In both cases the status return is SS$_ABORT. Note that the modem registers are not cleared.

The clean function is not supported in the DDCMP mode of operation.

2.4.3.6 Set Tributary Mode

The set tributary mode function either starts a tributary or modifies an existing one. The driver creates a circuit data block for a particular unit of the DMP11 device with the specified tributary address. The set tributary function must be performed before any communication can occur with the attached unit.

Because the DMF32 driver deals with only one tributary, the set tributary function starts both the tributary and the protocol. The data block describing the tributary has already been created.

The operating system provides the following combinations of function code and modifier:

- IO$_SETMODE—Modify tributary characteristics
- IO$_SETCHAR—Modify tributary characteristics
- IO$_SETMODE!IO$M_STARTUP—Start tributary
- IO$_SETCHAR!IO$M_STARTUP—Start tributary

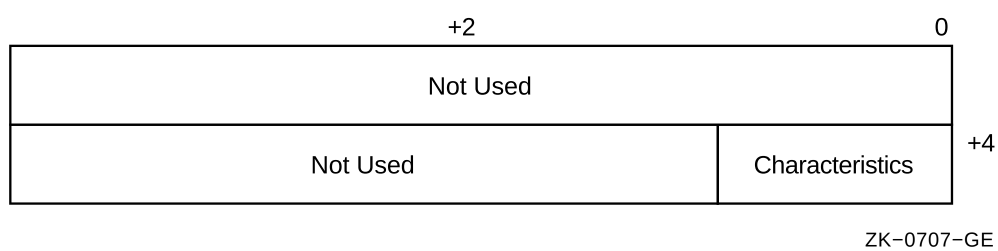
These codes take the following device- or function-dependent arguments:

- P1—The virtual address of a quadword characteristics buffer (optional)
- P2—The address of a descriptor for an extended characteristics buffer (optional)

Figure 2–5 shows the format of the P1 characteristics buffer. The following characteristic can be set in the second longword:

- XM$V_CHR_MOP—Set tributary to DDCMP maintenance mode

Figure 2–5 P1 Characteristics Buffer (Set Tributary)



The P2 buffer consists of a series of six-byte entries. The first longword contains the parameter identifier (ID), and the next longword contains one of the values that can be associated with the parameter ID. Figure 2–4 shows the format for this buffer.

Table 2–10 lists the parameter IDs and values that can be specified in the P2 buffer.

Table 2–10 P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCCI_TRI	Tributary address. Because the maximum physical address that the DMP11 or DMF32 can recognize is 255, only the first byte is actually used. For the DMP11, this parameter must be set before the tributary is started, unless the controller was set to run in point-to-point or DMC-compatible mode. For the DMF32, the tributary address always defaults to 1. Accepted values are 1 to 255.
NMA$C_PCCI_MRB[1]	Maximum number of buffers allocated from common pool for receive messages; 255 indicates unlimited number (default is unlimited). Accepted values are 1 to 255.
NMA$C_PCCI_MST[1]	Maintenance state. The following values can be specified: **Value** — **Meaning** NMA$C_STATE_ON — On NMA$C_STATE_OFF — Off (default)
NMA$C_PCCI_POL[1,2]	Latch polling state. The following values can be specified: **Value** — **Meaning** NMA$C_CIRPST_AUT — Automatic (default) NMA$C_CIRPST_ACT — Active NMA$C_CIRPST_INA — Inactive NMA$C_CIRPST_DIE — Dying NMA$C_CIRPST_DED — Dead
NMA$C_PCCI_TRT[1,2]	Transmit delay timer (default = 0).
NMA$C_PCCI_ACB[1,2]	Initial poll priority for active state of tributary (default = 255).
NMA$C_PCCI_ACI[1,2]	Rate of priority incrementing for active state of tributary (default = 0).
NMA$C_PCCI_IAB[1,2]	Initial poll priority for inactive state of tributary (default = 0).
NMA$C_PCCI_IAI[1,2]	Rate of priority incrementing for inactive state of tributary (default = 64).
NMA$C_PCCI_DYB[1,2]	Initial poll priority for dying state of tributary (default = 0).

[1]Only for the DMP11.

[2]A tributary-specific polling parameter. All timer values must be specified in milliseconds.

(continued on next page)

Table 2–10 (Cont.) P2 Extended Characteristics Values

Parameter ID	Meaning
NMA$C_PCCI_DYI[1,2]	Rate of priority incrementing for dying state of tributary (default = 16).
NMA$C_PCCI_IAT[1,2]	Number of no data message responses before changing state to inactive (default = 8).
NMA$C_PCCI_DYT[1,2]	Number of no responses before changing state to dying (default = 2).
NMA$C_PCCI_DTH[1,2]	Number of no responses before changing state to dead (default = 16).
NMA$C_PCCI_MTR[2]	Maximum number of abutting data messages that will be transmitted before deselecting the tributary (default = 4).
NMA$C_PCCI_BBT[1,2]	Timer value for tributary to indicate maximum amount of time for a selected tributary to transmit. If this value is exceeded, the tributary is babbling (default = 6000).
NMA$C_PCCI_RTT[2]	Retransmit timer for full-duplex point-to-point mode and selection timer for multipoint control and half-duplex point-to-point mode (default = 3000).

[1]Only for the DMP11.

[2]A tributary-specific polling parameter. All timer values must be specified in milliseconds.

If both P1 and P2 characteristics are specified, the P2 characteristics supersede the P1 characteristics. For example, if P1 specifies XM$M_CHR_MOP and P2 specifies NMA$C_PCCI_MST with a value of NMA$C_STATE_OFF, the tributary is in the normal DDCMP or data mode.

On receipt of the QIO request, the DMP11 driver verifies that a tributary address has been specified and determines whether this address is currently in use. If the address is in use, the tributary is not restarted. However, modifications to the tributary state or polling parameters are performed. If the tributary does not already exist, a new tributary is started.

On receipt of the QIO request to a DMF32, the driver modifies the tributary parameters and starts the protocol. The tributary state and the protocol state are equal. The driver does not verify that a tributary address has been provided. If an address has not been provided, it defaults to 1.

2.4.3.7 Shutdown Controller

The shutdown controller function shuts down the controller and disables the modem line. On completion of a shutdown controller request, all tributaries have been halted (including those tributaries not explicitly halted), all tributary buffers returned, and the controller reinitialized. For the DMF32, this function halts the tributary, the protocol, and the line. The controller cannot be used again until another IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP or IO$_SETCHAR!IO$M_CTRL!IO$M_STARTUP request has been issued (see Section 2.4.3.1).

The operating system provides the following combinations of function code and modifier:

- IO$_SETMODE!IO$M_CTRL!IO$M_SHUTDOWN—Shutdown controller
- IO$_SETCHAR!IO$M_CTRL!IO$M_SHUTDOWN—Shutdown controller

The shutdown controller function takes no device- or function-dependent arguments.

2.4.3.8 Shutdown Tributary

The shutdown tributary function halts, but does not delete, the specified tributary. On completion of a shutdown tributary request, the tributary is halted, all buffers are returned, and all pending I/O requests and received messages are aborted. Although the tributary cannot be used again until another IO$_SETMODE!IO$M_STARTUP or IO$_SETCHAR!IO$M_STARTUP request has been issued (see Section 2.4.3.6), all previously defined tributary parameters remain set (applicable only to the DMP11). For the DMF32, this function halts the tributary and the protocol. The attached device cannot be used until the tributary is restarted.

The operating system provides the following combinations of function code and modifier:

- IO$_SETMODE!IO$M_SHUTDOWN—Shutdown tributary
- IO$_SETCHAR!IO$M_SHUTDOWN—Shutdown tributary

The shutdown tributary function takes no device- or function-dependent arguments.

2.4.3.9 Enable Attention AST

The enable attention AST function requests that an attention AST be delivered to the requesting process when a status change occurs on the specified tributary. An AST is queued when the driver sets or clears either an error summary bit or any of the unit status bits (see Tables 2–3 and 2–4), or when a message is available and there is no waiting read request. The enable attention AST function is legal at any time, regardless of the condition of the unit status bits.

The operating system provides the following combinations of function code and modifier:

- IO$_SETMODE!IO$M_ATTNAST—Enable attention AST
- IO$_SETCHAR!IO$M_ATTNAST—Enable attention AST

These codes take the following device- or function-dependent arguments:

- P1—The address of an AST service routine or 0 for disable
- P2—Ignored
- P3—Access mode to deliver AST

The enable attention AST function enables an attention AST to be delivered to the requesting process once only. After the AST occurs, it must be explicitly reenabled by the function before the AST can occur again. The function is also subject to AST quotas.

The AST service routine is called with an argument list. The first argument is the current value of the second longword of the I/O status block (see Section 2.5). The access mode specified by P3 is maximized with the requester's access mode.

2.4.4 Sense Mode

The sense mode function returns the controller or tributary characteristics in the specified buffers.

The operating system provides the following function codes:

- IO$_SENSEMODE!IO$M_CTRL—Read controller characteristics
- IO$_SENSEMODE—Read tributary characteristics

These codes take the following device- or function-dependent arguments:

- P1—The address of a two-longword buffer into which the device characteristics are stored (optional). Figure 2–3 shows the characteristics buffer for controllers; Figure 2–5 shows the characteristics buffer for tributaries.
- P2—The address of a descriptor for a buffer into which the extended characteristics buffer is stored (optional). Figure 2–4 shows the format of the extended characteristics buffer.

All characteristics that fit into the buffer specified by P2 are returned. However, if all the characteristics cannot be stored in the buffer, the I/O status block returns the status SS$_BUFFEROVF. The second word of the I/O status block returns the size (in bytes) of the extended characteristics buffer returned by P2 (see Section 2.5).

2.4.4.1 Read Internal Counters

The read internal counters (IO$M_RD_COUNTS) subfunction reads the DDCMP internal counters. The operating system provides the following combinations of function codes and modifiers:

- IO$_SENSEMODE!IO$M_RD_COUNTS—Read tributary counters (DDCMP only)
- IO$_SENSEMODE!IO$M_CLR_COUNTS—Clears tributary counters (DDCMP only)
- IO$_SENSEMODE!IO$M_RD_COUNTS!IO$M_CLR_COUNTS—Read and then clear tributary counters (DDCMP only)
- IO$_SENSEMODE!IO$M_CTRL!IO$M_RD_COUNTS—Read controller counters (DDCMP and LAPB only)
- IO$_SENSEMODE!IO$M_CTRL!IO$M_CLR_COUNTS—Clear controller counters (DDCMP and LAPB only)
- IO$_SENSEMODE!IO$M_CTRL!IO$M_RD_COUNTS!IO$M_CLR_COUNTS—Read and then clear controller counters (DDCMP and LAPB only)

These codes take the following device- or function dependent arguments:

- P1—Ignored.
- P2—The address of a buffer descriptor into which the counters will be returned (Figure 2–6 shows the format of the buffer). Table 2–11 lists the parameter IDs that can be returned for DDCMP controllers, Table 2–12 lists parameter IDs that can be returned for LAPB controllers, and Table 2–13 lists the parameter IDs that can be returned for tributaries.

All counters that fit into the buffer specified by P2 are returned. However, if all the counters cannot be stored in the buffer, the I/O status block returns the status SS$_BUFFEROVF. The second word of the I/O status block returns the size, in bytes, of the extended characteristics buffer returned (see Section 2.5).

Figure 2–6 P2 Extended Characteristics Buffer (Sense Mode)

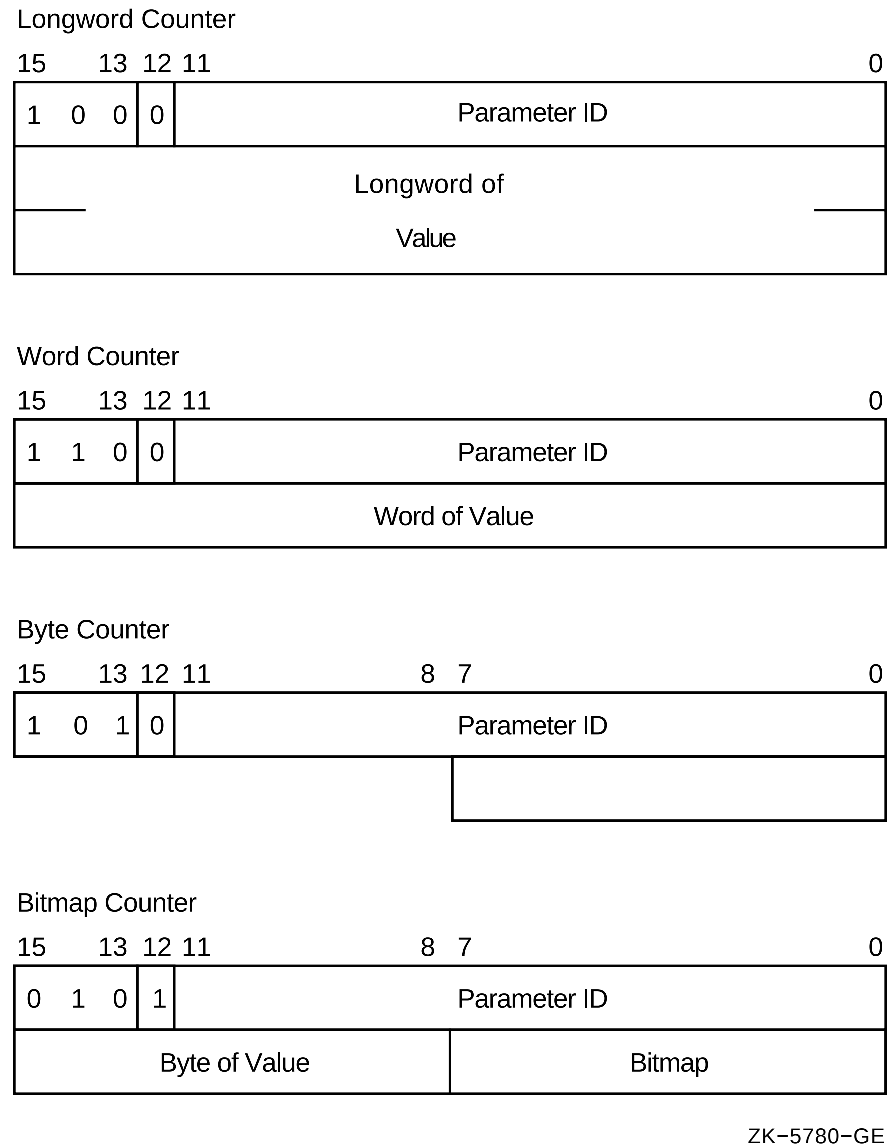


Table 2–11 DDCMP Controller Counter Parameter IDs

Parameter ID	Meaning
NMA$C_CTLIN_LPE	Number of local station errors bitmap counter.

Value	Meaning
1	Receive overrun SNAK set.
2	Receive overrun SNAK not set.
4	Transmitter underrun.
8	Message format error.

Parameter ID	Meaning
NMA$C_CTLIN_RPE	Number of remote station errors bitmap counter.

Value	Meaning
1	NAKs received due to receiver overrun.
2	NAKs received due to message format error.
4	SNAK set message format error.
8	Streaming tributary.

Table 2–12 LAPB Controller Counter Parameter IDs

Parameter ID	Meaning
NMA$C_CTCIR_DEI	Data errors inbound.

Table 2–13 Tributary Counter Parameter IDs

Parameter ID	Meaning
NMA$C_CTCIR_BRC	Number of bytes received by this station.
NMA$C_CTCIR_BSN	Number of bytes transmitted by this station.
NMA$C_CTCIR_DBR	Number of messages received by this station.
NMA$C_CTCIR_DBS	Number of messages transmitted by this station.
NMA$C_CTCIR_SIE	Number of selection intervals elapsed.
NMA$C_CTCIR_RBE	Remote buffer error bitmap counters.

Value	Meaning
1	Remote buffer unavailable.
2	Remote buffer too small.

(continued on next page)

Table 2–13 (Cont.) Tributary Counter Parameter IDs

Parameter ID	Meaning
NMA$C_CTCIR_LBE	Local buffer error bitmap counters.

Value	Meaning
1	Local buffer unavailable.
2	Local buffer too small.

Parameter ID	Meaning
NMA$C_CTCIR_SLT	Selection timeout bitmap counters.

Value	Meaning
1	No attempt to respond was made.
2	Attempt was made, but timeout still occurs.

Parameter ID	Meaning
NMA$C_CTCIR_RRT	Number of SACK settings when REP received.
NMA$C_CTCIR_LRT	Number of SREP settings.
NMA$C_CTCIR_DEI	Data error inbound bitmap counters.

Value	Meaning
1	NAK transmitted header CRC error.
2	NAK transmitted data CRC error.
4	NAK transmitted REP response.

Parameter ID	Meaning
NMA$C_CTCIR_DEO	Data error outbound bitmap counters.

Value	Meaning
1	NAK received header CRC error.
2	NAK received data CRC error.
4	NAK received REP response.

2.4.5 Diagnostic Support

The DMP11 and DMF32 drivers provide special capabilities for diagnostic support. The sections that follow describe these capabilities.

If a diagnostic buffer (P6) is specified with a physical I/O request, the eight one-byte device registers are dumped into it on completion of the request. (The DMF32 returns five one-word device registers.) The *DMP11 Technical Manual* and the *DMF32 Technical Manual* specify the contents of these registers. The P6 buffer does not return error counters.

2.4.5.1 Set Line Unit Modem Status

The set line unit modem status function sets the DMP11's line unit modem register. It is not supported for the DMF32. The operating system provides the following combinations of function code and modifier:

- IO$_SETMODE!IO$M_SET_MODEM—Set line unit modem status
- IO$_SETCHAR!IO$M_SET_MODEM—Set line unit modem status

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

- P1—The address of a longword buffer that contains new modem status. One or more of the symbolic offsets listed in the following table can be set in the buffer.

Offset	Meaning
XM$V_MDM_STNDBY	Select standby used with EIA modems
XM$V_MDM_MAINT2	Maintenance mode 2 for remote loopback
XM$V_MDM_MAINT1	Maintenance mode 1 for local loopback
XM$V_MDM_FREQ	Select frequency
XM$V_MDM_RDY	Data terminal ready to receive or transmit data
XM$V_MDM_POLL	Select polling modem mode

2.4.5.2 Read Line Unit Modem Status

The read line unit modem status function reads the DMP11's line unit modem register. The operating system provides the following combinations of function code and modifier:

- IO$_SENSEMODE!IO$M_RD_MODEM—Read line unit modem status
- IO$_SENSEMODE!IO$M_CTRL!IO$M_RD_MODEM—Read line unit modem status (DMF32)

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

- P1—The address of a longword buffer into which the line unit's modem status is stored. One or more of the bits listed in the following table can be set in the buffer.

Bit	Meaning
XM$V_MDM_CARRDET[1]	Receiver is active (Carrier Detect)
XM$V_MDM_MSTNDBY	STANDBY indication from modem
XM$V_MDM_CTS[1]	Data can be transmitted (CTS)
XM$V_MDM_DSR[1]	Modem is in service (DSR)
XM$V_MDM_HDX	Line unit is set to half-duplex mode
XM$V_MDM_RTS[1]	Request to send data from USART (RTS)
XM$V_MDM_DTR[1]	Line unit is available and on line (DTR)
XM$V_MDM_RING[1]	Modem has just been dialed up (RING)
XM$V_MDM_MODTEST	Modem is in TEST MODE

[1]Only for the DMF32

Bit	Meaning
XM$V_MDM_SIGQUAL	SIGNAL QUALITY from modem interface
XM$V_MDM_SIGRATE	SIGNAL RATE from modem interface

2.4.5.3 Read Device Status Slot

The read device status slot function reads a particular one-word memory location in a global or specified tributary status slot in the DMP11 controller. It is not supported for the DMF32. The operating system provides the following combinations of function code and modifier:

- IO$_SENSEMODE!IO$M_RD_MEM!IO$M_CTRL—Read global status slot
- IO$_SENSEMODE!IO$M_RD_MEM—Read tributary status slot

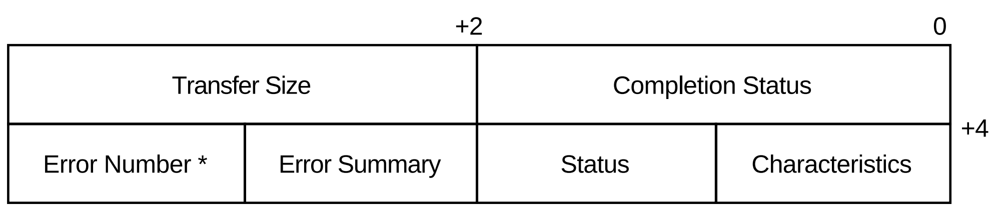
These codes take the following device- or function-dependent arguments:

- P1—The address of a longword buffer where the status slot information is stored
- P2—The tributary status slot address (0–31)

2.5 I/O Status Block

The I/O status block (IOSB) for all DMP11 and DMF32 functions is shown in Figure 2–7. Appendix A lists the completion status returns for these functions. (The OpenVMS system messages documentation provides explanations and suggested user actions for these returns.)

Figure 2–7 IOSB Contents for DMP11 and DMF32 Functions



* Only for DMP11

ZK−0708−GE

The first longword of the IOSB returns, in addition to the completion status, either the size (in bytes) of the data transfer or the size (in bytes) of the extended characteristics buffer returned by a sense mode function. The second longword returns the unit characteristics listed in Table 2–2; the line status bits listed in Table 2–3; the error summary bits listed in Table 2–4; and, for the DMP11, the total number of errors accrued.

2.6 Programming Example

The following sample program (Example 2–1) shows the typical use of QIO functions in DMP11 and DMF32 driver operations such as starting the controller and tributary and transmitting and receiving data.

To run this sample program on the first DMP11 in the system, enter the initial DCL command, ASSIGN XDA0: DEV.

Example 2–1 DMP11/DMF32 Program Example

```
        $ ASSIGN XDA0: DEV
        .TITLE  EXAMPLE - DMP11/DMF32  Device Driver Sample Program
        .IDENT  'X00'
        $IODEF                     ; Define I/O functions and modes
        $NMADEF                    ; Define Network Management symbols
        $XMDEF                     ; Define driver status flags

;
;  Macro definitions
;
        .macro  type      string,?l             ;
        store   <string>                        ;
        movl    #$$.tmpx,cmdorab+rab$l_rbf      ;
        movw    #$$.tmpx1,cmdorab+rab$w_rsz     ;
        $put    rab=cmdorab                     ;
        blbs    r0,1                            ;
        $exit_s                                 ;
l:                                              ;
        .endm   type                            ;

        .macro  store     string,pre
        .save
        .psect  $$$dev
        $$.tmpx=.
        pre
        .ascii  %string%
        $$.tmpx1=.-$$.tmpx
        .restore
        .endm   store

CMDOFAB:        $FAB    fac=put,fnm=sys$output:,- ; Output FAB
                        mrs=132,rat=cr,rfm=var
CMDORAB:        $RAB    ubf=cmdbuf,usz=cmdbsz,- ; Output RAB
                        fab=cmdofab
CMDBUF::        .BLKB   256                     ; Command buffer
CMDBSZ=         .-CMDBUF                        ; Buffer size
FAOBUFDSC:      .LONG   CMDBSZ,CMDBUF           ; FAO buffer
                                                ; descriptor
FAOLEN:         .BLKL   1                       ; FAO output buffer
                                                ; length
P2BUF::         .BLKL   50                      ; P2 buffer
P2BUFSZ=        .-P2BUF                         ; P2 buffer size
P2BUFDSC:       .LONG   P2BUFSZ,P2BUF           ; P2 buffer descriptor
P1BUF::         .BLKQ   1                       ; P1 buffer

P1BUFSZ=        .-P1BUF                         ; P1 buffer size
CHNL::          .BLKL   1                       ; Channel number
IOSB::          .BLKL   1                       ; I/O status block
DEVDSC:         .ASCID 'DEV'                    ; Device to assign
QIOREQDSC:      .LONG   QIOREQSZ,QIOREQ         ; QIO request status
QIOREQ:         .ASCII  'QIO completion status = !XL'
                .ASCII  'IOSB1 = !XL, IOSB2 = !XL'
QIOREQSZ=       .-QIOREQ                        ; Size of QIO status
                                                ; report
XMTBUFLEN=512                                   ; Size of transmit
                                                ; buffer
XMTBUF:         .REPEAT XMTBUFLEN
                .BYTE   ^X93                    ; Transmit data
                .ENDR
RCVBUF:         .BLKB   XMTBUFLEN
```

(continued on next page)

Example 2–1 (Cont.) DMP11/DMF32 Program Example

```
;
;  This is the start of the program section
;
START:: .WORD     0
        $OPEN     FAB=CMDOFAB                    ; Open output
        BLBC      R0,EXIT                        ;
        $CONNECT  RAB=CMDORAB                    ; Connect to output
        BLBC      R0,EXIT                        ;
        BRB       CONT                           ; Continue
EXIT:   $EXIT_S                                  ; Exit program

CONT:   TYPE      <DMP11/DMF32 Test Program>
        TYPE      <>
        $ASSIGN_S DEVNAM=DEVDSC,CHAN=CHNL        ; Assign unit
        BLBC      R0,EXIT                        ; Exit on error

;
;  Initialize and start controller
;
        MOVZWL    #XM$M_CHR_LOOPB!XM$M_CHR_DMC,P1BUF+4 ; Set P1 flags,
                                                 ; loopback and DMC
                                                 ; compatible
        MOVW      #XMTBUFLEN,P1BUF+2             ; Set P1 buffer size
        CLRL      P2BUFDSC                       ; Set zero length P2
                                                 ; buffer
        BSBW      INIT                           ; Issue QIO
;
;  Establish and start tributary
;
        CLRQ      P1BUF                          ; Reset P1 buffer
        MOVAB     P2BUF,R7                       ; Get address of P2
                                                 ; buffer
        MOVW      #NMA$C_PCCI_TRI,(R7)+          ; Set parameter code
        MOVZBL    #1,(R7)+                       ; Store trib address

        MOVZBL    #6,P2BUFDSC                    ; Store length of P2
                                                 ; buffer
        BSBW      ESTAB                          ; Issue QIO
;
;  Loopback data
;
        MOVZWL    #100,R9                        ; Loop device 100
                                                 ; times
10$:    BSBW      XMIT                           ; Issue transmit
        BSBW      RECV                           ; Issue receive
        MOVAB     XMTBUF,R1                      ; Get address of
                                                 ; transmit data
        MOVAB     RCVBUF,R2                      ; Get address of
                                                 ; received data
        MOVZWL    #XMTBUFLEN,R3                  ; Get number of bytes
                                                 ; to verify
20$:    CMPB      (R1)+,(R2)+                    ; Check data
        BNEQ      30$                            ;
        SOBGTR    R3,20$                         ;
        SOBGTR    R9,10$                         ;
        BRW       EXIT                           ; Exit
```

(continued on next page)

Example 2–1 (Cont.) DMP11/DMF32 Program Example

```
30$     TYPE      <*** Loopback buffer comparison error ***>
        BRW       EXIT                          ; Exit
;
;  Initialize controller QIO
;
INIT:   TYPE      <*** Initialize controller QIO ***>
        $QIO_S    chan=chnl,func=#io$_setchar!io$m_ctrl!io$m_startup,-
                  p1=p1buf,p2=#p2bufdsc,iosb=iosb,p3=#5
        BRW       QIO_STATUS                    ;

;
;  Start tributary QIO
;
ESTAB:  TYPE      <*** Startup tributary QIO ***>
        $QIO_S    chan=chnl,func=#io$_setchar!io$m_startup,-
                  p1=p1buf,p2=#p2bufdsc,iosb=iosb
        BRW       QIO_STATUS
;
;  Transmit data QIO
;
XMIT:   TYPE      <*** Transmit buffer QIO ***>
        $QIO_S    chan=chnl,func=#io$_writevblk,p1=xmtbuf,-
                  p2=#xmtbuflen,iosb=iosb
        BRW       QIO_XMTST
;
;  Receive data QIO
;
RECV:   TYPE      <*** Receive buffer QIO ***>
        $QIO_S    chan=chnl,efn=#2,func=#io$_readvblk,p1=rcvbuf,-
                  p2=#xmtbuflen,iosb=iosb
        .BRB      qio_status
        .ENABL    LSB
QIO_STATUS:                                     ; Check status of QIO
        BLBC      IOSB,10$                      ; Br if error on QIO
QIO_XMTST:                                      ; Check status of XMIT
        BLBC      R0,10$                        ; Br if error on
        RSB                                     ; request, else return
                                                ; to caller

10$     MOVZWL    IOSB,R1                       ; Get I/O status block
        PUSHL     R1                            ; Push I/O status block
        PUSHL     R0                            ; Push system service
                                                ; status
        PUSHAQ    FAOBUFDSC                     ; Push address of FAO
                                                ; buffer descriptor
        PUSHAW    FAOLEN                        ; Push address of
                                                ; output length
        PUSHAQ    QIOREQDSC                     ; Push address of
                                                ; input string
        CALLS     #5,@#SYS$FAO                  ; Get error message
        MOVAB     CMDBUF,CMDORAB+RAB$L_RBF      ; Get output buffer
                                                ; address
        MOVW      FAOLEN,CMDORAB+RAB$W_RSZ      ; Get output buffer
                                                ; length
        $PUT      CMDORAB                       ; Print error test
        BRW       EXIT                          ; Exit
        .DSABL    LSB

        .END      START
```

3

DR11–W and DRV11–WA Interface Driver

This chapter describes the use of the DR11–W interface driver (XADRIVER) in an OpenVMS VAX environment. The DRV11–WA uses the same driver; thus, unless otherwise stated, references to the DR11–W also apply to the DRV11–WA.

3.1 Supported Devices

The DR11–W is a general-purpose, 16-bit, parallel, direct-memory-access (DMA) data interface. It is capable of being used either as an interface between memory and a user device or as an interprocessor link (not DECnet) between two systems.

Because user devices of different or unknown capability can be connected to the interface that the XADRIVER presents, XADRIVER might be either insufficient or significantly inefficient for the application. For this reason, Digital provides limited support for the DR11–W and DRV11–WA when connected to foreign devices and provides the source code for XADRIVER in the OpenVMS VAX operating system distribution kit as a template for adding additional functionality.

Note that the driver is not supported if modifications are made to the source program. Digital strongly recommends that any modifications to device drivers be attempted only by those who are extremely familiar with the internal operation of the operating system. For additional information, refer to the *DR11–W Direct Memory Interface Module User's Guide*, the *DRV11–WA General Purpose DMA Interface User's Guide*, and the *OpenVMS VAX Device Support Manual*.

The DRV11–WA is similar to the DR11–W. However, it operates as an interface device that uses the 22-bit Q–bus rather than the UNIBUS. Unless otherwise indicated, the DRV11–WA driver performs the same QIO functions as the DR11–W driver; descriptions of DR11–W features also apply to the DRV11–WA. The DRV11–WA driver is supported for the MicroVAX II, but not the MicroVAX I.

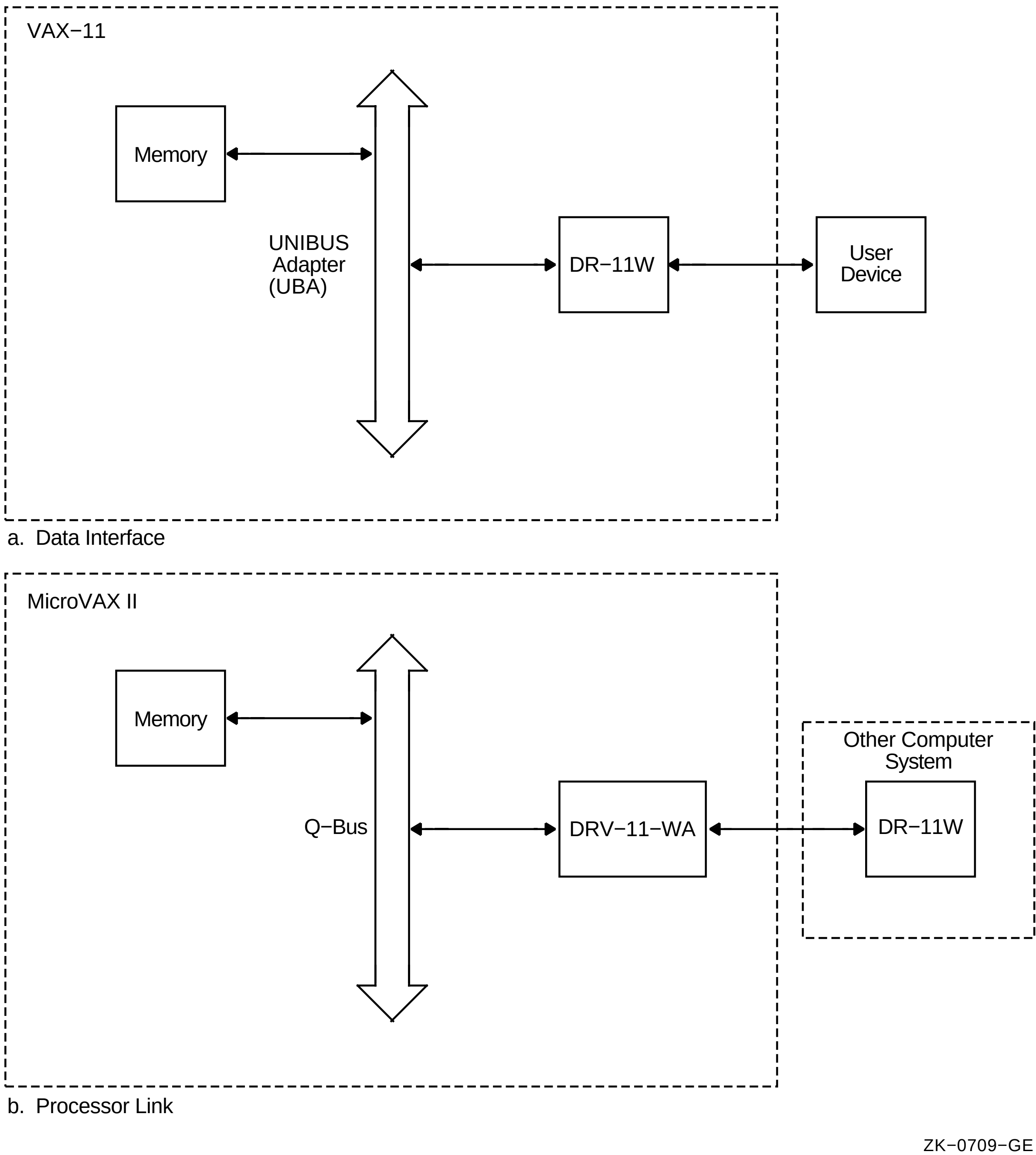
Note

Etch Revision Level E boards must be configured to be compatible with earlier versions of the DRV11–WA by installing jumpers W2, W3, and W6. These restrictions do not apply to the DR11–W.

You can link a DR11–W to another DR11–W, a DRV11–WA to another DRV11–WA, or a DR11–W to a DRV11–WA. The operating system does not support interprocessor links. You must write the code for any interprocessor communications operations.

Figure 3–1 shows two typical applications of the DR11–W and DRV11–WA.

Figure 3–1 Typical DR11–W/DRV11–WA Device Configurations



The driver (XADRIVER) allows general access to the features provided by the DR11–W and DRV11–WA devices. Function codes and modifiers are provided to control, and to transfer data between, the user device and the OpenVMS operating system.

3.1.1 Device Differences

The following differences between the DR11–W and the DRV11–WA affect the user at the QIO interface level; the referenced sections contain additional information about these differences:

- Unsolicited interrupts—The DRV11–WA driver does not acknowledge unsolicited interrupts (see Section 3.3).
- IO$M_WORD function modifier—The DRV11–WA driver does not perform word mode transfers (see Section 3.3).
- CSR error bit—The DRV11–WA driver detects some, but not all, hardware errors detected by the DR11–W driver (see Section 3.1.6).
- Error information register (EIR)—The DRV11–WA does not have an EIR (see Section 3.1.6).
- IO$M_RESET function modifier—The DRV11–WA cannot be reset in the same way as the DR11–W (see Section 3.3).
- IO$M_DATAPATH function modifier—The IO$M_DATAPATH function modifier is ignored for the DRV11–WA driver (see Section 3.3.3.1).

3.1.2 DRV11–WA Installation

In addition to the two installation considerations described in this section, follow the instructions in the hardware documentation when installing the DRV11–WA.

3.1.2.1 Type of Addressing

Bit 10 of the vector address selection switch is not used as part of the vector; it selects 18- or 22-bit addressing. Set the device to 22-bit addressing.

3.1.2.2 Device Address and Interrupt Vector Address Selection

Because the DRV11–WA is designed to be compatible with the DR11–B, the hardware documentation instructs you to set the device address and the interrupt vector address to those reserved for the DR11–B. However, the DRV11–WA is treated as much as possible like a DR11–W. Set the device address and interrupt vector address to those reserved for the DR11–W. (Set the device address to rank 19 and the interrupt vector address to rank 40, both in floating address space.)

Use the OpenVMS System Generation utility (SYSGEN) CONFIGURE command to calculate exact addresses. If you want to set up the device at the DR11–B address as described in the hardware documentation, configure the device using the following commands:

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> CONNECT GKpd0u /NOADAPTER
SYSGEN> LOAD SYS$SYSTEM:XADRIVER
SYSGEN> CONNECT XAA0 /ADAP=UB0/CSR=%O772410/VECTOR=%O124
SYSGEN> EXIT
```

3.1.3 DR11–W and DRV11–WA Transfer Modes

The DR11–W transfers data in block mode and in word mode. (Word-mode transfers are not supported with the DRV11–WA.) In block mode, all transfers are provided by the DMA facility. Each QIO request moves a single buffer of data between the user device and physical memory. One interrupt is generated on completion of the transfer. The transfer rate and transfer direction are controlled by the user device.

In block mode, the two types of UNIBUS or Q–bus transfers are single cycle and burst. During single-cycle transfers the bus is arbitrated for each word (two bytes) of information exchanged. Both the DR11–W and the DRV11–WA have a single cycle mode supported by the operating system.

Burst transfers result in the exchange of multiple words without arbitration of the bus. Two classes of burst mode transfers are possible, depending on the position of a switch on the module. On the DR11–W, the operating system only permits the use of dual cycle mode (class 1) in which two words are transferred for each arbitration of the UNIBUS. On the DRV11–WA, the operating system only permits the use of the 4-cycle mode in which four words are transferred for each arbitration of the Q–bus. Use burst mode transfers with caution. They can provide greater performance, but can prevent use of the bus by other devices for what might be unacceptable periods. Both the DR11–W and the DRV11–WA also have an N-cycle burst mode that cannot be used on OpenVMS VAX systems. On DRV11–WA boards prior to CS Revision Level B and Etch Revision Level D, N-cycle is the only form of burst mode available, and there is no burst mode selection switch on the module.

In word mode, a single QIO request transfers a buffer of data, with an interrupt requested for each word. Word mode is usually used to exchange control information between the application program and the user device. Once the proper control information has been accepted, a block-mode transfer can be started to exchange data.

In both block- and word-mode transfers, the transfer size is indicated by the byte count value specified in the P2 argument. The DR11–W and DRV11–WA transfer information between main memory and the user device in one-word (two-byte) units; transfers are counted on a word-by-word basis. However, the operating system counts information one byte at a time. Consequently, if the desired DR11–W or DRV11–WA transfer is 100 words, the P2 argument must specify 200 (bytes) for the transfer count value. If an odd number of bytes is specified for the transfer count, the driver rejects the QIO request.

Transfers to and from memory typically occur from sequentially increasing addresses. The user device can inhibit the increment to the next address.

During block-mode transfers, the user device controls the transfer direction through signals exchanged with the driver. Neither the operating system nor the application program has any control over the transfer direction. Consequently, a read or write request to the driver by the application program should be by convention, according to the intended action. An effect of this, regardless of whether a read or write QIO function is specified, is that the application program's data buffer is always checked for modify access (rather than read or write access) during block-mode transfers. In word mode, the transfer direction is controlled explicitly by the device driver.

Note

The meaning of the terms read and write can be misunderstood when discussing data transfers. This manual uses these terms for the application procedure running under the OpenVMS VAX operating system. A read operation involves the transfer of information from the user device to VAX memory. A write operation involves the transfer of information from VAX memory to the user device. Receive and input are synonymous with read operations; transmit and output are synonymous with write operations.

3.1.4 DR11–W and DRV11–WA Control and Status Register Functions

For each buffer of data transferred, the DR11–W or DRV11–WA driver allows for the exchange of an additional six bits of information: the function (FNCT) and status (STATUS) bits, which are included in the control and status register (CSR). These bits are accessible to an application process through the device driver QIO interface. The FNCT bits are labeled FNCT 1, FNCT 2, and FNCT 3. The STATUS bits are labeled STATUS A, STATUS B, and STATUS C.

The user device interfaced to the DR11–W or DRV11–WA interprets the value of the three FNCT bits. The QIO request that initiates the transfer specifies the IO$M_SETFNCT modifier to indicate a change in the value for the FNCT bits. The P4 argument of the request specifies this value. P4 bits 0 through 2 correspond to FNCT bits 1 through 3, respectively. Bits 3 through 31 are not used. If required, the FNCT bits must be set for each request. The FNCT bits set in the CSR are passed directly to the user device.

The DR11–W and DRV11–WA STATUS bits are available in bits 9 through 11 of the CSR, which correspond to STATUS bits C, B, A, respectively. On completion of all transfers, the STATUS bits are returned from the user device through the DR11–W or DRV11–WA to the IOSB. Neither the operating system nor the DR11–W/ DRV11–WA modifies these bits in any way. Thus, both FNCT and STATUS fields are defined solely by the user device. Except when used as an interprocessor link, the DR11–W or DRV11–WA takes no special action based on the state of these fields, and the FNCT bits remain set until explicitly changed with the IO$M_ SETFNCT function modifier.

The DR11–W and DRV11–WA CSR STATUS bits should not be confused with the status values returned in the I/O status block.

The function modifier IO$M_CYCLE sets the CSR CYCLE bit for the transfer specified by the QIO request. In block mode, the CYCLE bit initiates the transfer of the first word of data. In word mode, IO$M_CYCLE has no effect.

Section 3.1.7 describes the special meaning given to the FNCT and STATUS bits by the DR11–W or DRV11–WA hardware and device driver when used as an interprocessor link.

3.1.5 Data Registers

Two registers are used to transfer information to and from the user device. The input data register (IDR) contains the last data value transferred into the DR11–W or DRV11–WA from the user device. The output data register (ODR) contains the last value transferred from the DR11–W or DRV11–WA to the user device. During block mode operations, these registers are controlled automatically and require no explicit action on the part of the application program. During word-mode write operations, the DR11–W driver loads the ODR with each successive data word; each word is then available to the user device. During word-mode read operations, the driver reads the IDR and stores the value in memory. Interrupts from the DR11–W synchronize reading and writing the IDR and ODR when in word mode.

3.1.6 Error Reporting

The error information register (EIR) is used for reporting certain error conditions to the application program at the completion of each request. As the result of a user device action, the device sets the ATTN bit in the CSR. The CSR ERROR bit is also set at this time. If ERROR is set during a block-mode transfer, the transfer is aborted. Table 3–5 in Section 3.4 lists the EIR and CSR bit assignments for the I/O status block.

The DRV11–WA detects some, but not all, types of errors detected by the DR11–W. Specifically, the error bit in the CSR (bit 15) for the DRV11–WA signals attention interrupts, nonexistent memory errors, and power failures at the remote device, but does not signal multicycle request errors or parity errors. The DRV11–WA does not have an EIR. The driver always returns zeros in place of the EIR in the fourth word of the IOSB when an I/O operation is completed.

3.1.7 Link Mode of Operation

The XADRIVER driver can control two DR11–Ws, two DRV11–WAs, or a DR11–W and a DRV11–WA connected as interprocessor links between two computer systems.

Note

The DRV11–WA to DRV11–WA link mode of operation is not possible with earlier board versions. Digital does not support the DRV11–WA to DRV11–WA link mode of operation.

Control switches on the DR11–W and DRV11–WA modules are set to place the hardware in the link mode configuration. You must set these switches and use either the set mode or the set characteristics function to instruct the driver to function in link mode.

In link operations, two cooperating processes exchange data through the devices, which function as a memory-to-memory interface. This feature requires that the two processes agree on, and establish a basis for describing, the direction of the data transfer, the message sizes, and arbitrating use of the link.

In link operations, the FNCT and STATUS bits are given special meaning by the DR11–W or DRV11–WA hardware and the device driver. Proper operation of the DR11–W or DRV11–WA as an interprocessor link depends on the correct use of these bits. The driver does not enforce correct use of the FNCT and STATUS bits. When issuing a QIO request to the DR11–W or DRV11–WA in link mode with IO$M_SETFNCT specified, the correct values and sequence of FNCT bits must be provided by the application image. Table 3–1 lists the FNCT and STATUS bits and what actions occur when the DR11–W or DRV11–WA is in link mode. Table 3–5 lists the CSR bit assignments.

Table 3–1 Control and Status Register FNCT and STATUS Bits (Link Mode)

Bit	Function
FNCT 1	Indicates whether the DR11–W or DRV11–WA at this end of the link is to transmit or receive data. If FNCT 1 is 0, the DR11–W or DRV11–WA transmits data from memory to the associated DR11–W or DRV11–WA at the other end of the link. If FNCT 1 is 1, the DR11–W or DRV11–WA receives data from the associated DR11–W or DRV11–WA and stores it in memory. (Note that two DRV11–WAs cannot be linked together.) For proper operation, one system must set FNCT 1 to 1 (for receive) and the associated system must set FNCT 1 to 0 (for transmit).
FNCT 2	Interrupts the remote processor. For proper operation, the driver must be set to operate as a link. When a set mode or set characteristics function is used to instruct the driver to perform a link operation, the driver does not leave FNCT 2 set. Instead, the driver sets and then immediately clears the bit to provide a pulse, rather than a level, to the associated system.
FNCT 3	Indicates whether the nonprocessor request (NPR) transfers that follow occur as single-cycle or burst-mode transfers. If FNCT 3 is 0, burst transfers are performed. If FNCT 3 is 1, single-cycle transfers are performed. Note that burst-mode transfers can occupy the UNIBUS or Q–bus for long periods, to the exclusion of other devices on the same bus.
STATUS A	Returns the value of FNCT 3 set in the associated computer system. When an interrupt is returned from the associated computer denoting the need to exchange a message, STATUS A indicates whether the request that follows is to be set up for single-cycle or for burst operation.
STATUS B	Returns the value of FNCT 2 set in the associated system. Because the DR11–W driver, when configured as a link, never leaves FNCT 2 set, STATUS B is never read as a 1. When STATUS B is set, ATTENTION and, in turn ERROR, are set in the DR11–W or DRV11–WA. When the driver handles the resulting interrupt, it attempts to clear ATTENTION. If ATTENTION cannot be cleared, it indicates that the condition causing it was a level, held true by the associated system. Since ATTENTION can be set by conditions other than FNCT 2, for example, the error ACLO in the associated system, treating FNCT 2 as a pulse allows the receiving DR11–W to differentiate between an error and a normal processor interrupt request.
STATUS C	Returns the value of the FNCT 1 bit sent by the associated computer. STATUS C indicates whether the DMA transfer that follows is a transmit or a receive operation.

If a DR11–W in link configuration sets one or more of the three CSR FNCT bits, the other DR11–W will perform one or more of the following actions:

- Request an interrupt
- Specify the intended transfer direction for a block-mode transfer that follows
- Declare whether the transfer is to take place in burst or single-cycle operation

In each case, the value written into the FNCT bits of the first DR11–W is available and is read from the STATUS bits of the other DR11–W.

Because either process can initiate the data transfer, arbitration for the use of the link is automatic. If both processes want to write or both want to read, a timeout occurs. A timeout also occurs if either process neglects to specify the agreed-upon transfer direction or message size. Each process should specify a different timeout period or a different time before re-requesting the link after a timeout. These actions, which preclude a lockup of the link, are not enforced by the driver.

If an attention interrupt is generated, it indicates that either the DR11–W or DRV11–WA associated with the other system is initiating a transfer or that the other DR11–W or DRV11–WA is going off line because of a power failure. The DR11–W driver's ability to clear ATTENTION (see the description of STATUS B in Table 3–1) allows a data transfer to be distinguished from a hardware error. If an error occurs and ATTENTION can be cleared, SS$_DRVERR is returned as the status. If ATTENTION cannot be cleared, SS$_CTRLERR is returned.

3.2 Device Information

You obtain information about DR11–W or DRV11–WA characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *OpenVMS System Services Reference Manual*.)

$GETDVI returns DR11–W- or DRV11–WA-specific characteristics when you specify the item codes DVI$_DEVCHAR and DVI$_DEVDEPEND. Tables 3–2 and 3–3 list these characteristics. The $DEVDEF macro defines the device-independent characteristics; the $XADEF macro defines the device-dependent characteristics.

Table 3–2 DR11–W and DRV11–WA Device-Independent Characteristics

Characteristic[1]	Meaning
Dynamic Bits (Conditionally Set)	
DEV$M_AVL	Device is on line and available.
DEV$M_ELG	Error logging is enabled for this device.
Static Bits (Always Set)	
DEV$M_IDV	Input device.
DEV$M_ODV	Output device.
DEV$M_RTM	Real-time device.

[1]Defined by the $DEVDEF macro

Table 3–3 DR11–W and DRV11–WA Device-Dependent Characteristics

Value[1]	Meaning
XA$M_DATAPATH	Describes which UNIBUS adapter data path is in use. 0 = direct data path; 1 = buffered data path. The initial state of this bit is 0. (Not applicable to the DRV11–WA.)
XA$M_LINK	Describes whether the DR11–W or DRV11–WA is used as a link or as a user device interface. 0 = user device interface; 1 = link. The initial state of this bit is 0.

[1]Defined by the $XADEF macro

DVI$_DEVTYPE and DVI$_DEVCLASS return the device type and device class names, which are defined by the $DCDEF macro. The device type for the DR11–W is DT$_DR11W; the device type for the DRV11–WA is DT$_XA_DRV11WA.

The device class for both the DR11–W and DRV11–WA is DC$_REALTIME. DVI$_DEVBUFSIZ returns the default buffer size, which is 65,535.

3.3 DR11–W and DRV11–WA Function Codes

The XADRIVER can perform logical, virtual, and physical I/O operations. The basic I/O functions are read, write, set mode, and set characteristics. Table 3–4 lists these functions and their function codes. The following sections describe these functions in greater detail.

Table 3–4 DR11–W Function Codes

Function Code	Arguments	Type[1]	Function Modifiers	Function
IO$_READLBLK	P1,P2,P3,P4,P5	L	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Read logical block.
IO$_READVBLK	P1,P2,P3,P4,P5	V	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Read virtual block.
IO$_READPBLK	P1,P2,P3,P4,P5	P	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Read physical block.
IO$_WRITELBLK	P1,P2,P3,P4,P5	L	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Write logical block.
IO$_WRITEVBLK	P1,P2,P3,P4,P5	V	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Write virtual block.
IO$_WRITEPBLK	P1,P2,P3,P4,P5	P	IO$M_SETFNCT IO$M_WORD[2] IO$M_TIMED IO$M_CYCLE IO$M_RESET	Write physical block.
IO$_SETMODE	P1,P3	L	IO$M_ATTNAST	Set DR11–W or DRV11–WA characteristics for subsequent operations.
IO$_SETCHAR	P1,P3	P	IO$M_ATTNAST IO$M_DATAPATH	Set DR11–W or DRV11–WA characteristics for subsequent operations.

[1]V = virtual, L = logical, P = physical (there is no functional difference in these operations)

[2]Not applicable to the DRV11–WA

Although the XADRIVER does not differentiate among logical, virtual, and physical I/O functions (all are treated identically), you must have the required privilege to issue a request.

The read and write functions take the following device- or function-dependent arguments:

- P1—The starting virtual address of the buffer that is to receive data for a read operation or the virtual address of the buffer that is to send data to the DR11–W for a write operation. Modify access to the buffer, rather than read or write access, is checked for all block-mode read and write requests.
- P2—The size of the data buffer in bytes (the transfer count). Because the DR11–W performs word transfers, the transfer count must be an even value. The maximum transfer size is 65,534 bytes. If a larger number is specified, the high-order bits of this field are ignored.
- P3—The timeout period for this request (in seconds). The value specified must be equal to or greater than 2. IO$M_TIMED must be specified. The default timeout value for each request is 10 seconds.
- P4—The value of the DR11–W command and status register (CSR) function (FNCT) bits to be set. If IO$M_SETFNCT is specified, the low-order three bits of P4 (2:0) are written to the CSR FNCT bits 3:1 (respectively) at the time of the transfer.
- P5—The value (low two bytes) to be loaded into the DR11–W output data register (ODR). IO$M_SETFNCT must be specified and the transfer count (P2) must be 0.

If a direct data path (DDP) is used (see Section 3.3.3.1), the address specified by the P1 argument must be word-aligned. However, if a buffered data path (BDP) is used, byte alignment is allowed. All transfers through the BDP, which is available only on the UNIBUS, must occur from sequential, increasing addresses. If the user device interfaced to the DR11–W cannot conform to this requirement, the DDP must be used.

The transfer count specified by the P2 argument must be an even number of bytes. If an odd number is specified, an error (SS$_BADPARAM) is returned in the I/O status block (IOSB). If the transfer count is 0, the driver will transfer no data. However, if IO$M_SETFNCT is specified and P2 is 0, the driver will set the FNCT bits in the DR11–W CSR, load the low two bytes specified in P5 into the DR11–W ODR, and return the current CSR status bit values in the IOSB.

The read and write functions can take the following function modifiers:

- IO$M_SETFNCT—Sets the FNCT bits in the DR11–W CSR before the data transfer is initiated. The low-order three bits of the P4 argument specify the FNCT bits. The user device that interfaces with the DR11–W or DRV11–WA receives the FNCT bits directly, and their value is interpreted entirely by the device.

  Additionally, if the transfer count (P2) is 0, load the value specified in P5 into the device ODR.

  If a link operation is specified in the device-dependent characteristics (XA$M_ LINK = 1), FNCT 2 will not be left set (that is, it will be set and immediately cleared) in the device CSR.
- IO$M_WORD—Performs the data transfer in word mode rather than in DMA block mode (not applicable to the DRV11–WA). In word mode an interrupt occurs for each word transferred. This allows the exchange of a small amount of data to establish the parameters for a block-mode data transfer that follows.

If IO$M_WORD is included in a write request, the first word in a user's buffer is loaded into the DR11–W ODR. The driver then waits for an interrupt before proceeding to load the next word or complete the request. If IO$M_WORD is included in a read request, the driver waits for an interrupt and then reads a word from the DR11–W IDR and stores it in the user's buffer.

Interrupts are initiated when either the user device or, when in link operation, the associated DR11–W sets ATTENTION.

If the DR11–W or DRV11–WA receives an unsolicited interrupt, no read or write request is posted. If the next request is for a word-mode read, the driver returns the word read from the DR11–W IDR and stores it in the first word of the user's buffer. In this case the driver does not wait for an interrupt.

The DRV11–WA does not respond to unsolicited interrupts from a remote device; the DRV11–WA only acknowledges interrupts when a DMA transfer is outstanding. Consequently, word-mode transfers are not possible on a DRV11–WA because the device does not acknowledge the interrupt that occurs when the I/O operation is completed; the QIO waits indefinitely or times out. (In some cases, you can work around this problem by causing the remote device to generate an interrupt, which makes the local DRV11–WA complete the I/O operation with an SS$_OPINCOMPL status.)

- IO$M_TIMED—Uses the timeout value in the P3 argument rather than the default timeout value of 10 seconds.
- IO$M_CYCLE—Sets the cycle bit in the DR11–W or DRV11–WA CSR for this request. In block mode, this initiates the first NPR cycle. For user devices, the application of the cycle bit is dependent on the specific device. In word mode, IO$M_CYCLE is ignored. In link operations, only the transmitting DR11–W or DRV11–WA must set CYCLE and then only after the companion DR11–W has its receive request initiated.
- IO$M_RESET—Performs a device reset to the DR11–W before any I/O operation is initiated. This function does not affect any other device on the system.

  The DRV11–WA can be reset only by initializing the Q–bus and all other devices attached to the Q–bus. Therefore, when the IO$M_RESET function modifier is used to reset the DRV11–WA, the XADRIVER simulates a reset by setting the word count register (WCR) to indicate one word left to be transferred and setting the CYCLE bit to complete the transfer. If the driver is not performing a transfer at the time of a reset, the reset is a no-op.

On completion of each read or write request, including those requests with a zero transfer count, the current value of the DR11–W or DRV11–WA CSR and DR11–W EIR is returned in the I/O status block.

3.3.1 Read

Read functions provide for the direct transfer of data from the user device that interfaces with the DR11–W or DRV11–WA into the user process's virtual memory address space. The operating system provides the following function codes:

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

Five function-dependent arguments and five function modifiers are used with these codes. These arguments and modifiers are described at the beginning of Section 3.3.

3.3.2 Write

Write functions provide for the direct transfer of data to the user device that interfaces with the DR11–W or DRV11–WA from the user process's virtual memory address space. The operating system provides the following function codes:

- IO$_WRITELBLK—Write logical block
- IO$_WRITEVBLK—Write virtual block
- IO$_WRITEPBLK—Write physical block

Five function-dependent arguments and five function modifiers are used with these codes. These arguments and modifiers are described at the beginning of Section 3.3.

3.3.3 Set Mode and Set Characteristics

Set mode operations affect the operation and characteristics of the associated DR11–W or DRV11–WA. The operating system defines two types of set mode functions: set mode and set characteristics. These functions allow the user process to set or change the device characteristics. The following function codes are provided:

- IO$_SETMODE—Set mode (no I/O privilege required)
- IO$_SETCHAR—Set characteristics (requires physical I/O privilege)

These functions take the following device- or function-dependent arguments:

- P1—The virtual address of a quadword characteristics buffer. If the function modifier IO$M_ATTNAST is specified, P1 is the address of the AST service routine. In this case, if P1 is 0, all attention ASTs are disabled.
- P3—The access mode to deliver the AST (maximized with the requester's access mode). If IO$M_ATTNAST is not specified, P3 is ignored.

Figure 3–2 shows the quadword P1 characteristics buffer for IO$_SETMODE and IO$_SETCHAR.

Figure 3–2 P1 Characteristics Buffer

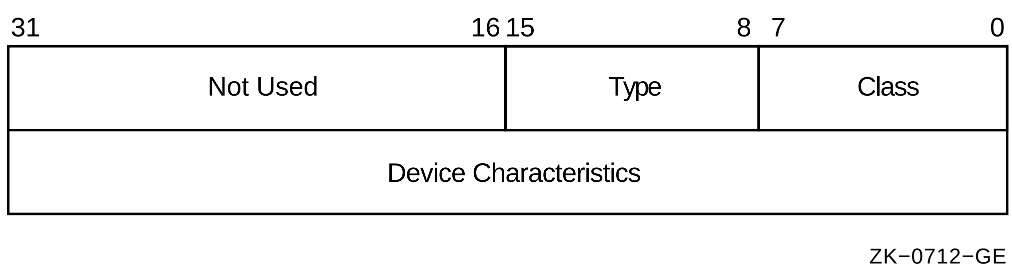


Table 3–3 lists the device characteristics for the set mode and set characteristics functions. The device class value must be DC$_REALTIME. The device type value must be DT$_DR11W or DT$_XA_DRV11WA. These values are defined by the $DCDEF macro.

3.3.3.1 Set Mode Function Modifiers

The IO$_SETMODE and IO$_SETCHAR function codes can take the following function modifier:

- IO$M_ATTNAST—Enable attention AST

This function modifier allows the user process to queue an attention AST for delivery when an asynchronous or unsolicited condition is detected by the DR11–W or DRV11–WA driver. Unlike ASTs for other QIO functions, use of this function modifier does not increment the I/O count for the requesting process or lock pages in memory for I/O buffers. Each AST is charged against the user's AST limit.

Attention ASTs are delivered when any of the following occur:

- Any block- or word-mode data transfer request is completed.
- An unsolicited interrupt from the DR11–W occurs. (The DRV11–WA does not respond to unsolicited interrupts.)
- An attention AST is queued and a previous unsolicited interrupt has not been acknowledged.
- A device timeout occurs.

The Cancel I/O on Channel ($CANCEL) system service is used to flush attention ASTs for a specific channel.

The enable attention AST function modifier enables an attention AST to be delivered to the requesting process once only. After the AST occurs, it must be explicitly reenabled by the function modifier before the AST can occur again. This function modifier does not update the device characteristics.

When the AST is delivered, the AST parameter contains the contents of the DR11–W or DRV11–WA CSR in the low two bytes and the value read from the DR11–W or DRV11–WA IDR in the high two bytes.

In addition to IO$M_ATTNAST, the IO$_SETCHAR function code can take the following function modifier:

- IO$M_DATAPATH—Use the data path specified by XA$M_DATAPATH in the P1 characteristics buffer

The IO$M_DATAPATH function modifier allows the user to specify either the direct data path (DDP) or a buffered data path (BDP) for block-mode transfers through the UNIBUS adapter.

The device-specific characteristic XA$M_DATAPATH is used to switch between use of the DDP and the BDP. If XA$M_DATAPATH is set, the BDP is used; if clear, the DDP is used. Regardless of the value of XA$M_DATAPATH, the choice of data path has no effect unless the function modifier IO$M_DATAPATH is also specified, which requires physical I/O privilege.

Note

Use caution when specifying data transfers through the BDP. The user device has access to several hardware functions: C0 and C1 inhibit word count increment and inhibit bus address increment. If these signals are used out of context of the expected UNIBUS adapter constraints for BDPs, the result is unpredictable.

Unlike the UNIBUS, the Q–bus does not provide a choice between a direct data path and a buffered data path; the IO$M_DATAPATH function modifier is ignored for the DRV11–WA.

3.4 I/O Status Block

The I/O status block (IOSB) for DR11–W or DRV11–WA read and write functions is shown in Figure 3–3. On completion of each read or write request, the I/O status block is filled with system and DR11–W or DRV11–WA status information.

Figure 3–3 IOSB Contents for DR11 and DRV11 Functions

+2	IOSB
Byte Count	Status
DR11−W EIR	DR11−W CSR

ZK−0713−GE

The first longword of the I/O status block contains I/O status returns and the byte count. Appendix A lists the status returns for read and write functions. (The OpenVMS system messages documentation provides explanations and suggested user actions for these returns.) The byte count is the actual number of bytes transferred by the request. If the request ends in an error, the byte count might differ from the requested number of bytes. If a power failure, timeout, or the Cancel I/O on Channel ($CANCEL) system service stops the request, the value in the byte count field is not valid.

The third and fourth words of the I/O status block contain the values of the DR11–W CSR and EIR on completion of the request. (The DRV11–WA has a CSR but not an EIR; the driver always returns zeros in the fourth word of the IOSB when an I/O operation is completed.) Table 3–5 lists the bit assignments for these two words. The *DR11–W User's Manual* provides additional information about the EIR and CSR.

Table 3–5 EIR and CSR Bit Assignments

Word	Bit	Function
EIR	0	Register flag
	1–7	(not applicable)
	8	N-cycle burst
	9	Burst timeout (sets ERROR)
	10	PARITY (sets ERROR)
	11	ACLO (sets ERROR)
	12	Multicycle request (sets ERROR)
	13	ATTENTION (sets ERROR)

(continued on next page)

Table 3–5 (Cont.) EIR and CSR Bit Assignments

Word	Bit	Function
	14	Nonexistent memory (sets ERROR)
	15	ERROR (generates interrupt when set)
CSR	0	GO
	1	FNCT 1
	2	FNCT 2
	3	FNCT 3
	4	Extended bus address 16
	5	Extended bus address 17
	6	Interrupt enable
	7	READY
	8	CYCLE
	9	STATUS C
	10	STATUS B
	11	STATUS A
	12	Maintenance mode
	13	ATTENTION (sets ERROR)
	14	Nonexistent memory (sets ERROR)
	15	ERROR (generates interrupt when set)

3.5 Programming Example

A sample program residing in the SYS$EXAMPLES directory demonstrates how to perform transfers across a DR11–W to DRV11–WA or a DR11–W to DR11–W interprocessor link. The sample program includes the following modules:

- XALINK.MAR—Places the device in link mode
- XAMESSAGE.MAR—Performs the actual transfer of data
- XATEST.FOR—Solicits parameters for the transfer from the user and calls the XALINK.MAR and XAMESSAGE.MAR modules
- XATEST.COM—Compiles and links the sample program

Example 3–1, which consists of the module XAMESSAGE.MAR, shows how an actual memory-to-memory link might be implemented using the XADRIVER. All actions are invoked through the $QIO interface by a nonprivileged image.

Note

XAMESSAGE.MAR is a demonstration program, not an application. The program may not work in all circumstances. See the template warning at the beginning of Example 3–1.

XAMESSAGE.MAR includes the following features:

- Either system can function as the transmitter or the receiver. For any given exchange, one system must be the transmitter and one must be the receiver.
- Either the transmitter or the receiver can call XAMESSAGE first, which is made possible by the driver's ability to keep track of unsolicited attention interrupts. XAMESSAGE uses this feature for the following reasons:
  - To synchronize the DMA exchange
  - To ensure that the receiver issues the block-mode read request first
  - To ensure that the transmitter sets the CYCLE bit to initiate the first NPR transfer
- If either the transmitter or receiver specifies unequal transfer sizes or does not match the transfer direction, either a timeout occurs or one of the procedures returns an error. The caller must resolve these discrepancies.

Table 3–6 lists the main flow of the program. Note that paths for transmit and receive and for DR11–W and DRV11–WA are combined in the same module (XAMESSAGE).

The three parts of Table 3–6 describe the operation of XAMESSAGE in three different device configurations:

- A DRV11–WA transmitting a message to a DR11–W
- A DR11–W transmitting a message to a DRV11–WA
- A DR11–W transmitting a message to another DR11–W

The two right-hand columns describe the action taken by each device involved in the transfer. The leftmost column contains the name of the routine in XAMESSAGE that performs the respective action: MAIN refers to the main routine for XAMESSAGE, AST_GO refers to the AST routine by that name, AST_COM refers to the AST routine called AST_COMPLETION, and ASYNC means that the action occurs asynchronously and is not controlled directly by any code in XAMESSAGE.

Table 3–6 XAMESSAGE Program Flow

DRV11–WA (Transmitter) to DR11–W (Receiver)

XAMESSAGE	DRV11–WA (Transmitter)	DR11–W (Receiver)
MAIN	Issue block-mode read request.	Enable attention AST.
AST_GO		Execute attention AST as a result of interrupt from transmitter.
AST_GO		Issue block-mode read request.
AST_GO	Complete block-mode read request prematurely as a result of the interrupt at the beginning of the receiver's read request.	
AST_GO	Issue block-mode write request.	
ASYNC	Perform DMA transfer.	Perform DMA transfer.

(continued on next page)

Table 3–6 (Cont.) XAMESSAGE Program Flow

DRV11–WA (Transmitter) to DR11–W (Receiver)

XAMESSAGE	DRV11–WA (Transmitter)	DR11–W (Receiver)
AST_COM	Execute completion AST and check for errors.	Execute completion AST and check for errors.

DR11–W (Transmitter) to DRV11–WA (Receiver)

XAMESSAGE	DRV11–WA (Receiver)	DR11–W (Transmitter)
MAIN	Issue block-mode read.	Enable attention AST.
AST_GO		Execute attention AST as a result of interrupt from receiver.
AST_GO		Issue block-mode write request.
ASYNC	Perform DMA transfer.	Perform DMA transfer.
AST_COM	Execute completion AST and check for errors.	Execute completion AST and check for errors.

DR11–W (Transmitter) to DR11–W (Receiver)

XAMESSAGE	DR11–W (Receiver)	DR11–W (Transmitter)
MAIN	Enable attention AST.	Enable attention AST.
MAIN		Momentarily set the FNCT 2 bit via a 0-length transfer to interrupt the receiver.
AST_GO	Execute attention AST as a result of interrupt from transmitter.	
AST_GO	Issue block-mode read request.	
AST_GO		Execute attention AST as a result of interrupt from receiver.
AST_GO		Issue block-mode write request.
ASYNC	Perform DMA transfer.	Perform DMA transfer.
AST_COM	Execute completion AST and check for errors.	Execute completion AST and check for errors.

Example 3–1 DR11–W/DRV11–WA Program Example (XAMESSAGE.MAR)

```
        .TITLE          XAMESSAGE
        .IDENT          'V04-001'
```

(continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11–WA Program Example (XAMESSAGE.MAR)

```
;*************************************************************************
;*  DIGITAL ASSUMES  NO  RESPONSIBILITY TO SUPPORT THE SOFTWARE DESCRIBED   *
;*  IN THIS MODULE, NOR TO ANSWER INQUIRIES ABOUT IT.                       *
;*                                                                          *
;*  THIS SOFTWARE MODULE IS PART OF A TEMPLATE WHICH MAY REQUIRE CUSTOMER   *
;*  MODIFICATIONS TO WORK IN ALL CIRCUMSTANCES.                             *
;*************************************************************************
;++
; ABSTRACT:
;
;       This module allows you to connect a DR11-W to a DRV11-WA; or
;       a DR11-W to another DR11-W in an interprocessor link and to
;       perform data transfers from one processor to the other.
;--

        .SBTTL        LOCAL DEFINITIONS AND STORAGE
;++
; XAMESSAGE ROUTINE
;
; CALLING SEQUENCE:
;
;       CALL (BUFFER_ADDRESS,BUFFER_SIZE,TRANSFER_DIRECTION,CHANNEL,-
;             EVENT_FLAG,TIME_OUT,STATUS_ADDRESS,LOCAL_DEVICE,REMOTE_DEVICE)
;
;               BUFFER_ADDRESS = ADDRESS OF DATA BUFFER TO TRANSFER
;               BUFFER_SIZE = SIZE IN BYTES OF DATA BUFFER TO TRANSFER.
;                 NOTE THAT RECEIVER AND TRANSMITTER MUST AGREE ON THE
;                 SIZE OF THE TRANSFER.
;               TRANSFER_DIRECTION = DIRECTION FOR DATA TO GO
;                       0 = TRANSMIT
;                       1 = RECEIVE
;               CHANNEL = CHANNEL ASSIGNED TO DEVICE (DR11-W OR DRV11-WA)
;               EVENT_FLAG = EVENT FLAG TO SET WHEN TRANSFER COMPLETE
;               TIME_OUT = I/O TIME-OUT VALUE IN SECONDS
;               STATUS_ADDRESS = ADDRESS OF 20 BYTE ARRAY TO RECEIVE
;                 FINAL STATUS - ONLY FILLED IN IF USER'S PARAMETERS ARE
;                 ALL VALID.
;                       IOSB - 8 BYTES
;                               I/O STATUS BLOCK FROM QUEUE I/O REQUEST
;                       ERROR - 4 BYTES - NOT USED - FOR COMPATIBILITY
;                               WITH OLD VERSIONS OF THIS MODULE.
;                       STATE - 4 BYTES
;                               THIS FIELD TRACKS WHICH QIO WAS THE LATEST
;                               ONE TO BE PERFORMED.
;                               01 - LAST QIO WAS ONE IN THE MAIN ROUTINE.
;                               02 - LAST QIO WAS ONE IN AST_GO.
;                       SSRV_STS - 4 BYTES
;                               VALUE OF R0 RETURNED FROM THE LAST SYSTEM
;                               SERVICE EXECUTED.
;               LOCAL_DEVICE = TYPE OF DEVICE AT LOCAL END OF LINK.
;                       DR11_W = 1
;                       DRV11_WA = 2
;               REMOTE_DEVICE = TYPE OF DEVICE AT REMOTE END OF LINK.
;                       DR11_W = 1
;                       DRV11_WA = 2
;--
```

(continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11–WA Program Example (XAMESSAGE.MAR)

```
        $SSDEF
; PARAMETER OFFSETS.
BUFFER_P = 4
BUF_SIZE_P = 8
DIRECTION_P = 12
CHAN_P = 16
EFN_P = 20
TIME_P = 24
STS_ADDR_P = 28
LCL_DEVICE_P = 32
REM_DEVICE_P = 36
        .PSECT        XADATA,LONG

; SAVED PARAMETER VALUES.
BUFFER:          .LONG       0            ; SAVED BUFFER ADDRESS
BUF_SIZE:        .LONG       0            ; SAVED BUFFER SIZE
DIRECTION:       .LONG       0            ; DIRECTION OF TRANSFER
CHAN:            .LONG       0            ; SAVED CHANNEL ASSIGNED TO DR11-W
EFN:             .LONG       0            ; SAVED EVENT FLAG NUMBER
TIME:            .LONG       0            ; SAVED TIME-OUT VALUE
STS_ADDR:        .LONG       0            ;ADDRESS OF CALLERS STATUS VARIABLE

; DEFINE DEVICE TYPES AT BOTH ENDS OF INTERPROCESSOR LINK.

DR11_W = 1
DRV11_WA = 2
LCL_DEVICE:      .BLKL       1             ; TYPE OF DEVICE ON THIS SYSTEM.
REM_DEVICE:      .BLKL       1             ; TYPE OF DEVICE AT OTHER
                                           ;   END OF LINK.
AST:             .BLKL       1

; NOTE - ORDER IS ASSUMED FOR NEXT FOUR VARIABLES
IOSB:            .QUAD       0             ; QIO IOSB
ERROR:           .LONG       0             ; ERROR VALUE PARAMETER
STATE:           .LONG       0             ; STATE VARIABLE
SSRV_STS:        .LONG       0             ; SYSTEM SERVICE STATUS


        .PAGE
        .SBTTL    VALIDATE AND SAVE CALLER'S PARAMETERS
        .PSECT    XACODE,NOWRT
        .ENTRY    XAMESSAGE,^M<R2,R3,R4,R5>
```

(continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11–WA Program Example (XAMESSAGE.MAR)

```
; VALIDATE AND SAVE CALLER'S PARAMETERS
        CLRQ      W^IOSB                     ; CLEAR IOSB
        CLRL      W^ERROR                    ; CLEAR ERROR FIELD
        CLRL      W^SSRV_STS                 ; CLEAR SYS SERVICE RETURN STATUS
        CMPW      (AP),#9                    ; MUST HAVE 9 PARAMETERS
        BEQL      10$                        ; BR IF OKAY
        BRW       BADPARAM                   ; BR TO SIGNAL ERROR
10$:    MOVL      BUFFER_P(AP),W^BUFFER      ; GET BUFFER ADDRESS
        MOVL      @BUF_SIZE_P(AP),W^BUF_SIZE ; GET BUFFER SIZE
        BNEQ      20$                        ; BR IF OKAY
        BRW       BADPARAM                   ;TRANSFER SIZE IS NONZERO- ILLEGAL
20$:    MOVZBL    @DIRECTION_P(AP),W^DIRECTION ; GET TRANSFER DIRECTION FLAG
        CMPL      W^DIRECTION,#2             ; THE ONLY LEGAL VALUES ARE 0,1
        BLEQU     25$                        ; BR IF OKAY
        BRW       BADPARAM                   ; ELSE BR TO SIGNAL ERROR
25$:    MOVL      @CHAN_P(AP),W^CHAN         ; FETCH CHANNEL
        MOVL      @EFN_P(AP),W^EFN           ; AND EVENT FLAG
        BEQL      BADPARAM                   ; MUST SPECIFY EVENT FLAG
        MOVL      @TIME_P(AP),W^TIME         ; FETCH TIME-OUT VALUE
        BNEQ      30$                        ; IF NONZERO, USE IT.
        MOVZBL    #5,W^TIME                  ;ELSE USE SOME "REASONABLE" VALUE
30$:    MOVL      STS_ADDR_P(AP),W^STS_ADDR ; GET ADDRESS OF STATUS ARRAY
        BEQL      BADPARAM                   ; IF NOT SPECIFIED, ERROR
        CLRL      @W^STS_ADDR                ; INITIALIZE STATUS VALUE
        MOVZBL    @LCL_DEVICE_P(AP),W^LCL_DEVICE ; GET LOCAL DEVICE TYPE
        CMPL      #DRV11_WA,W^LCL_DEVICE     ; IS LOCAL DEVICE A DRV11-WA?
        BEQLU     35$                        ; BRANCH IF SO.
        CMPL      #DR11_W,W^LCL_DEVICE       ; IS LOCAL DEVICE A DR11-W?
        BNEQU     BADPARAM                   ; ERROR IF IT'S NOT EITHER.
35$:    MOVZBL    @REM_DEVICE_P(AP),W^REM_DEVICE ; GET REMOTE DEVICE TYPE
        CMPL      #DRV11_WA,W^REM_DEVICE     ; IS REMOTE DEVICE A DRV11-WA?
        BEQLU     50$                        ; BRANCH IF SO.
        CMPL      #DR11_W,W^REM_DEVICE       ; IS REMOTE DEVICE A DR11-W?
        BNEQU     BADPARAM                   ; ERROR IF IT'S NOT EITHER.
50$:    $CLREF_S  EFN=EFN                    ; MAKE SURE EFN IS CLEAR
        BLBS      R0,100$                    ; BR IF NO SYS SERVICE ERROR
        RET
100$:   CMPL      #DRV11_WA,W^LCL_DEVICE     ; DISPATCH BASED ON LOCAL
                                             ;   DEVICE TYPE
        BEQL      DRV11_WA_START             ; LOCAL DEVICE IS DRV11-WA
        BRW       DR11_W_START               ; LOCAL DEVICE IS DR11-W

BADPARAM:
        MOVZWL    #SS$_BADPARAM,R0           ; ELSE RETURN ERROR.
        RET


        .PAGE
        .SBTTL    START MESSAGE PROCESSOR
```

(continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11–WA Program Example (XAMESSAGE.MAR)

```
DRV11_WA_START:                                 ; THE LOCAL DEVICE IS A DRV11-WA
        BLBC      W^DIRECTION,10$               ; BRANCH IF IT'S A TRANSMIT
                                                ;   OPERATION
        MOVAL     W^AST_COMPLETION,W^AST        ; AST_COMPLETION IS THE AST FOR
                                                ;   RECEIVE
        BRB       20$
10$:    MOVAL     W^AST_GO,W^AST                ; AST_GO IS THE AST FOR TRANSMIT
                                                ;   OPERATION
20$:    MOVL      #01,W^STATE                   ; STATE = 1 => LAST QIO WAS IN
                                                ;   MAIN ROUTINE.
        $QIO_S    CHAN=W^CHAN,-                 ; BLOCK-MODE READ - EVEN IF IT'S
                  FUNC=#<IO$_READLBLK!IO$M_TIMED!IO$M_SETFNCT>,- ; TRANSMIT
                  IOSB=W^IOSB,-
                  ASTADR=@W^AST,-
                  P1=@W^BUFFER,-                ; ADDRESS OF CALLER'S DATA BUFFER
                  P2=W^BUF_SIZE,-               ; LENGTH OF DATA BUFFER
                  P3=W^TIME,-                   ; TIMEOUT VALUE
                  P4=#7                         ; INTERRUPT+READ
        BRW       MAIN_EXIT                     ; EXIT MAIN ROUTINE.
DR11_W_START:                                   ; LOCAL DEVICE IS DR11-W
        MOVL      #01,W^STATE                   ; STATE = 1 => LAST QIO WAS IN
                                                ;   MAIN ROUTINE.
        $QIO_S    CHAN=W^CHAN,-                 ; QIO TO ENABLE AST'S
                  FUNC=#<IO$_SETMODE!IO$M_ATTNAST>,-
                  IOSB=W^IOSB,-
                  P1=W^AST_GO
        BLBC      R0,MAIN_EXIT                  ; BRANCH ON ERROR - ALL DONE.
        BLBS      W^DIRECTION,MAIN_EXIT         ; BRANCH IF THIS IS A RECEIVE
                                                ;   OPERATION
        CMPL      #DR11_W,W^REM_DEVICE          ; IS REMOTE DEVICE A DR11-W?
        BNEQU     MAIN_EXIT                     ; BRANCH IF NOT.
        $QIO_S    CHAN=W^CHAN,-                 ; PERFORM 0-LENGTH QIO.  THIS
                  FUNC=#<IO$_WRITELBLK!IO$M_SETFNCT>,- ; SERVES TO SET THE
                  IOSB=W^IOSB,-                 ; FNCT BITS (CONTAINED IN P4),
                  P1=@W^BUFFER,-                ; IN THE CSR, INTERRUPTING THE
                                                ;   REMOTE DR11-W.
                  P2=#0,-
                  P4=#2
MAIN_EXIT:
        MOVL      R0,W^SSRV_STS                 ; SAVE QIO STATUS RETURN
        MOVC3     #20,W^IOSB,@W^STS_ADDR        ; RETURN STATUS TO THE USER
        BLBS      W^SSRV_STS,10$                ; IF SUCCESS, DON'T SET EVFLAG YET
        $SETEF_S  EFN=W^EFN                     ; IF ERROR, SET EVENT FLAG
                                                ;    -- ALL DONE.
10$:    MOVL      W^SSRV_STS,R0                 ; RESTORE R0 STATUS RETURN.
        RET


       .PAGE
       .SBTTL   AST_GO - AST THAT INITIATES THE QIO TO PERFORM ACTUAL TRANSFER

;
; This AST is called to perform the $QIO which begins the actual transfer
; of user data.  (Hence the name AST_GO.)
;
        BLBS     W^DIRECTION,AST_RECEIVE      ; BRANCH IF RECEIVE OPERATION
```

(continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11–WA Program Example (XAMESSAGE.MAR)

```
;
; On a DR11-W, this AST is delivered as a result of an interrupt from the
; remote device, so no status checking is necessary.  On a DRV11-WA, this
; AST is delivered as a result of an intentionally premature I/O completion,
; so we expect the status return to be SS$_OPINCOMPL.
;
AST_XMIT:
        CMPL     #DRV11_WA,W^LCL_DEVICE     ; IS LOCAL DEVICE A DRV11-WA?
        BNEQ     20$                        ; BRANCH IF NOT.
        CMPW     W^IOSB,#SS$_OPINCOMPL      ; STATUS SHOULD BE SS$_OPINCOMPL.
        BEQL     20$                        ; BR IF EXPECTED STATUS
        BRW      IO_DONE                    ; ELSE ERROR
20$:    MOVL     #02,W^STATE                ; STATE = 2 => LAST QIO WAS IN
                                            ;   AST_GO.
        $QIO_S   CHAN=W^CHAN,-              ; BLOCK-MODE WRITE
                 FUNC=#<IO$_WRITELBLK!IO$M_TIMED!IO$M_SETFNCT!IO$M_CYCLE>,-
                 IOSB=W^IOSB,-
                 ASTADR=W^AST_COMPLETION,-
                 P1=@W^BUFFER,-             ; ADDRESS OF CALLER'S DATA BUFFER
                 P2=W^BUF_SIZE,-            ; LENGTH OF BUFFER
                 P3=W^TIME,-                ; TIMEOUT VALUE
                 P4=#4                      ; FNCT BITS FOR CSR
        BLBS     R0,40$                     ; RETURN IF QIO STARTED OK
        BRW      IO_DONE                    ; ALL DONE IF ERROR OCCURRED.
40$:    RET                                 ; DISMISS THIS AST, AND
                                            ;   WAIT FOR AST_COMPLETION
;
; AST_RECEIVE is only used by the DR11-W, since the DRV11-WA initiates
; the actual data transfer from the main routine when it is the receiver.
;
AST_RECEIVE:
        MOVL     #02,W^STATE                ; STATE = 2 => LAST QIO WAS IN
                                            ;   AST_GO.
        $QIO_S   CHAN=W^CHAN,-              ; BLOCK-MODE READ
                 FUNC=#<IO$_READLBLK!IO$M_TIMED!IO$M_SETFNCT>,-
                 IOSB=W^IOSB,-
                 ASTADR=W^AST_COMPLETION,-  ; ADDRESS OF AST FOR I/O COMPLETION
                 P1=@W^BUFFER,-             ; ADDRESS OF CALLER'S DATA BUFFER
                 P2=W^BUF_SIZE,-            ; LENGTH OF DATA BUFFER
                 P3=W^TIME,-                ; TIMEOUT VALUE
                 P4=#7                      ; INTERRUPT+READ
        BLBS     R0,10$                     ; RETURN IF QIO STARTED OK
        BRW      IO_DONE                    ; ON ERROR, WE'RE ALL DONE.
10$:    RET
```

(continued on next page)

Example 3–1 (Cont.) DR11–W/DRV11–WA Program Example (XAMESSAGE.MAR)

```
        .PAGE
        .SBTTL   AST_COMPLETION - COMPLETION ROUTINE FOR I/O TRANSFER.
        .ENTRY   AST_COMPLETION,^M<R2,R3,R4,R5>
;
; This AST is called when the actual transfer of data is complete.  Note that
; the status value in the IOSB must be checked by the caller when we're done.
; IO_DONE is also called when an error occurs and the handshaking sequence
; must be terminated.
;
IO_DONE:
        MOVC3    #20,W^IOSB,@W^STS_ADDR      ; RETURN STATUS TO THE USER
        $SETEF_S EFN=W^EFN                   ; SET THE CALLER'S EVENT FLAG
        MOVZBL   #SS$_NORMAL,R0              ; SIGNAL SUCCESSFUL AST COMPLETION.
        RET
        .END
```

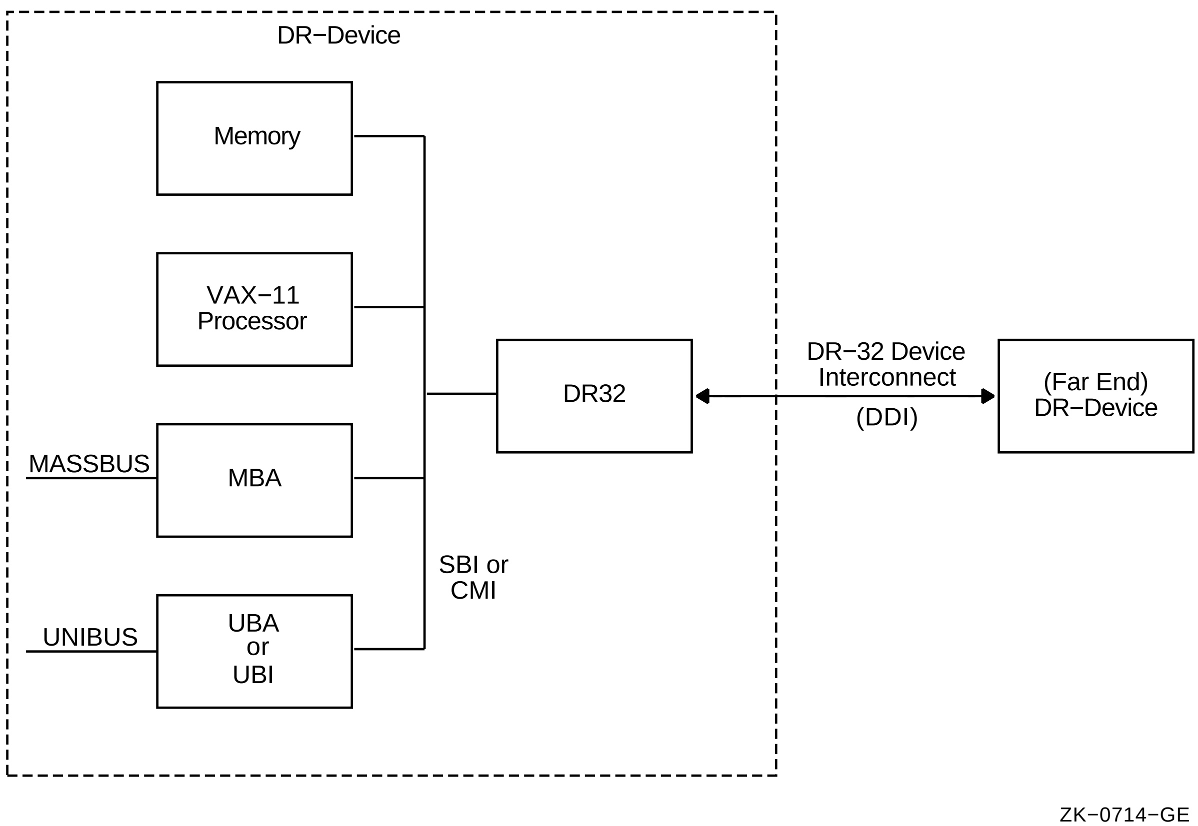
4
DR32 Interface Driver

This chapter describes the use of the DR32 interface driver in an Open VMS VAX environment.

4.1 Supported Device

The DR32 is an interface adapter that connects the internal memory bus of a VAX processor to a user-accessible bus called the DR32 device interconnect (DDI). Two DR32s can be connected to form a VAX processor-to-processor link (not DECnet). Figure 4–1 shows the relationship of the DR32 to an OpenVMS VAX system and the DR32 device interconnect (DDI).

Figure 4–1 Basic DR32 Configuration



As a general-purpose data port, the DR32 is capable of moving continuous streams of data to or from memory at high speed. Data from a user device to disk storage must go through an intermediate buffer in physical memory.

4.1.1 DR32 Device Interconnect

The DR32 device interconnect (DDI) is a bidirectional path for the transfer of data and control signals. Control signals sent over the DDI are asynchronous and interlocked; data transfers are synchronized with clock signals. Any connection to the DDI is called a **DR device**. The DDI provides a point-to-point connection between two DR devices, one of which must be a VAX processor. The DR device connected to the external end of the DDI is called the **far-end DR device**.

4.2 DR32 Features and Capabilities

The DR32 driver provides the following features and capabilities:

- 32-bit parallel data transfers
- High bandwidth (6 megabytes/second on the DDI with a VAX–11/780 or 3.12 megabytes/second on a VAX–11/750)
- Word or byte alignment of data
- Half-duplex operation
- Hardware-supported (I/O driver-independent) memory mapping
- Separate control and data interconnects
- Command and data chaining
- Direct software link between the DR32 and the user process
- Synchronization of the user program with DR32 data transfers
- Transfers initiated by an external device

The following sections describe command and data chaining, data transfers, power failure, and interrupts.

4.2.1 Command and Data Chaining

Command chaining is the execution of commands without software intervention for each command. Commands are chained in the sense that they follow each other on a queue. After a QIO function starts the DR32, any number of DR32 commands can be executed during that QIO operation. This process continues until either the transfer is halted (a command packet is fetched that specifies a halt command) or an error occurs. (Section 4.4.3 describes command packets.)

Command packets can specify data chaining. In data chaining, a number of physical memory buffers appear as one large buffer to the far-end DR device. Data chaining is completely transparent to this device; transfers are seen as a continuous stream of data. Chained buffers can be of arbitrary byte alignment and length. The length of a transfer appears to the far-end DR device as the total of all the byte counts in the chain, and because chains in the DR32 can be of unlimited length, the device interprets the byte count as potentially infinite.

4.2.2 Far-End DR Device-Initiated Transfers

For the far-end DR device, the DR32 provides the capability of initiating data transfers to memory (initiating random-access mode). This mode is used when two DR32s are connected to form a processor-to-processor link. Random access consists of data transfers to or from memory without notification of the VAX processor. Random access can be discontinued either by specifying a command packet with random access disabled or by an abort operation from either the controlling process or the far-end DR device.

4.2.3 Power Failure

If power fails on the DR32 interface but not on the system, the DR32 driver aborts the active data transfer and returns the status code SS$_POWERFAIL in the I/O status block. If a system power failure occurs, the DR32 driver completes the active data transfer when power is recovered and returns the status code SS$_POWERFAIL.

4.2.4 Interrupts

The DR32 interface can interrupt the DR32 driver for any of the following reasons:

- An abort has occurred. The QIO operation is completed.
- A power failure has occurred.
- The power has been turned on.
- An unsolicited control message has been sent to the DR32. If this command packet's interrupt control field is properly set up, a packet AST interrupt occurs. The interrupt occurs after the command packet obtained from the free queue (FREEQ) is placed on the termination queue (TERMQ).
- The DR32 enters the halt state. The QIO operation is completed.
- A command packet that specifies an unconditional interrupt has been placed onto TERMQ. The result is a packet AST.
- A command packet with the **interrupt when TERMQ empty** bit set was placed on an empty TERMQ. The result is a packet AST.

4.3 Device Information

You can obtain information about DR32 characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *OpenVMS System Services Reference Manual*.)

$GETDVI returns DR32 characteristics when you specify the item code DVI$_ DEVCHAR. Table 4–1 lists these characteristics, which are defined by the $DEVDEF macro.

Table 4–1 DR32 Device Characteristics

Characteristic[1]	Meaning
Dynamic Bit (Conditionally Set)	
DEV$M_AVL	Device is available.
Static Bits (Always Set)	
DEV$M_IDV	Input device.
DEV$M_ODV	Output device.

[1]Defined by the $DEVDEF macro

(continued on next page)

Table 4–1 (Cont.) DR32 Device Characteristics

Characteristic[1]	Meaning
	Static Bits (Always Set)
DEV$M_RTM	Real-time device.

[1]Defined by the $DEVDEF macro

DVI$_DEVTYPE and DVI$_DEVCLASS return the device type and class names, which are defined by the $DCDEF macro. The device type is DT$_DR780 for the DR780 and DT$_DR750 for the DR750. The device class for the DR32 is DC$_REALTIME. DVI$_DEVDEPEND returns a longword field in which the low-order byte contains the last data rate value loaded into the DR32 data rate register.

4.4 Programming Interface

The DR32 interface is supported by a device driver, a high-level language procedure library of support routines, and a program for microcode loading.

After issuing an IO$_STARTDATA request to the DR32 driver, application programs communicate directly with the DR32 interface by inserting command packets onto queues. This direct link between the application program and the DR32 interface provides faster communication by avoiding the necessity of going through the I/O driver.

Two interfaces are provided for accessing the DR32: a QIO interface and a support routine interface. The QIO interface requires that the application program build command packets and insert them onto the DR32 queues. The support routine interface, on the other hand, provides procedures for these functions and, in addition, performs housekeeping functions such as maintaining command memory.

The support routine interface was designed to be called from high-level languages, such as FORTRAN, where the data manipulation required by the QIO interface might be awkward. Note, however, that the user of the support routines interface must be as knowledgeable about the DR32 and the meaning of the fields in the command packets as the user of the QIO interface.

4.4.1 DR32—Application Program Interface

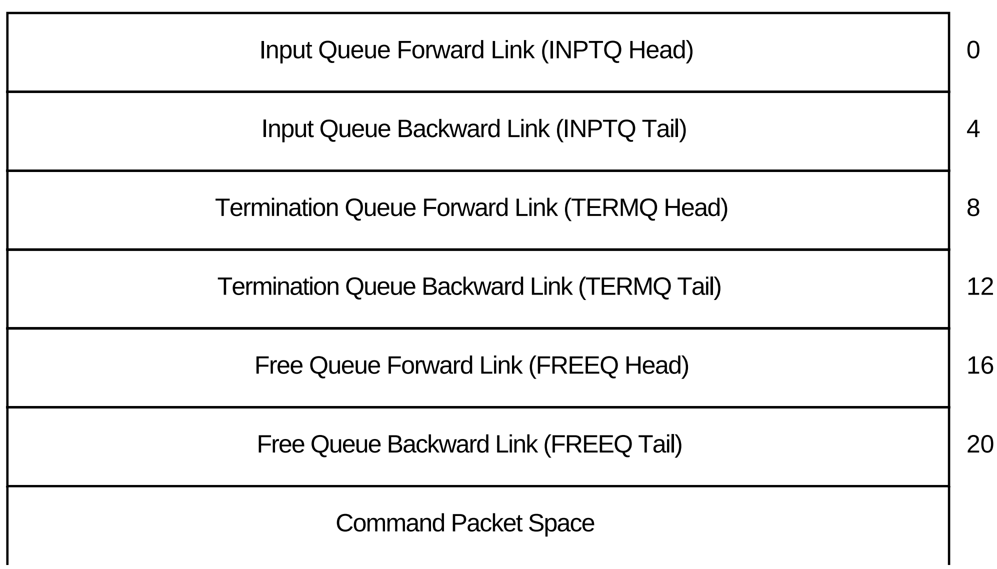
Application programs interface with the DR32 through two memory areas. These areas are called the **command block** and the **buffer block**. The addresses and sizes of the blocks are determined by the application program and are passed to the DR32 driver as arguments to the IO$_STARTDATA function, which starts the DR32 (see Section 4.4.5.2).

Both blocks are locked into memory while the DR32 is active. The buffer block defines the area of memory that is accessible to the DR32 for the transfer of data between the far-end DR device and the DR32. The command block contains the headers for the three queues that provide the communication path between the DR32 and the application program, and space in which to build command packets.

The interface between the DR32 and the application program contains three queues: the input queue (INPTQ), the termination queue (TERMQ), and the free queue (FREEQ). Information is transferred between the DR32 and the far-end DR device through command packets. The three queue structures control the flow of command packets to and from the DR32. The application program builds a command packet and inserts it onto INPTQ. The DR32 removes the packet, executes the specified command, enters some status information, and then inserts the packet onto TERMQ. Unsolicited input from the far-end DR device is placed in packets removed from FREEQ and inserted onto TERMQ.

The INPTQ, TERMQ, and FREEQ headers are located in the first six longwords of the command block. Because the queues are self-relative—meaning they use the VAX self-relative queue instructions (INSQHI, INSQTI, REMQHI, and REMQTI)—the headers must be quadword-aligned. The application program must initialize all queue headers. Figure 4–2 shows the position of the queue headers in the command block. Section 4.4.2 describes queue processing in greater detail.

Figure 4–2 Command Block (Queue Headers)



ZK−0716−GE

4.4.2 Queue Processing

Three queue structures control the flow of command packets to and from the DR32:

- Input queue (INPTQ)
- Termination queue (TERMQ)
- Free queue (FREEQ)

The DR32 removes command packets from the heads of FREEQ and INPTQ and inserts command packets onto the tail of TERMQ. For command sequences initiated by the application program, the DR32 removes command packets from the head of INPTQ, processes them, and returns them to the tail of TERMQ. Queue processing is performed by the DR32 with the equivalent of the INSQTI and REMQHI instructions. To remove a packet from INPTQ, the DR32 executes the equivalent of REMQHI HDR, CMDPTR where CMDPTR is a DR32 register used as a pointer to the current command packet and HDR specifies the INPTQ header. To insert a packet onto TERMQ, the DR32 executes the equivalent of INSQTI CMDPTR, HDR. The user process performs similar operations with the queues, inserting packets onto the head or tail of INPTQ and normally removing packets from the head of TERMQ.

If any of the queues are currently being accessed by the DR32, the program's interlocked queue instructions will fail for either of the following reasons:

- The DR32 is currently removing a packet from INPTQ or FREEQ, or inserting a packet onto TERMQ, and the operation will be completed shortly.
- The DR32 detects an error condition, such as an unaligned queue, that prevents it from completing the queue operation. In this case, the transfer is aborted and the I/O status block contains the error that caused the abort.

To distinguish between these two conditions, the application program must include a queue retry mechanism that retries the queue operation a reasonable number of times (for example, 25) before determining that an error condition exists. An example of a queue retry mechanism is shown in the DR32 queue I/O functions program example (in Section 4.7.2).

If the DR32 discerns that any of the queues are interlocked, it retries the operation until it completes or the DR32 is aborted.

4.4.2.1 Initiating Command Sequences

If a command packet is inserted onto an empty INPTQ, the application program must notify the DR32 of this event. This is done by setting bit 0, the GO bit, in a DR32 register. The IO$_STARTDATA function returns the GO bit's address to the application program. After notification by the GO bit that there are command packets on its INPTQ, the DR32 continues to process the packets until INPTQ is empty.

The GO bit can be safely set at any time. While processing command packets, the DR32 ignores the GO bit. If the GO bit is set when the DR32 is idle, the DR32 will attempt to remove a command packet from INPTQ. If INPTQ is empty at this time, the DR32 clears the GO bit and returns to the idle state.

4.4.2.2 Device-Initiated Command Sequences

If the DR device that interfaces the far end of the DDI is capable of transmitting unsolicited control messages, messages of this type can be transmitted to the local DR32. These messages are not synchronized to the application program command flow. Therefore, the DR32 uses a third queue, FREEQ, to handle unsolicited messages. Normally, the application program inserts a number of empty command packets onto FREEQ to allow the external device to transmit control messages.

If a control message is received from the far-end DR device, the DR32 removes an empty command packet from the head of FREEQ, fills the device message field of this packet with the control message and, when the transmission is completed, inserts the packet onto the tail of TERMQ. (The device message field in this command packet must be large enough for the entire message or a length error will occur.) The application program then removes the packet from TERMQ. If the command packet is from FREEQ, the XF$M_PKT_FREQPK bit in the DR32 status longword is set.

4.4.3 Command Packets

To provide for direct communication between the controlling process and the DR32, the DR32 fetches commands from user-constructed command packets located in physical memory. Command packets contain commands for the DR32, such as the direction of transfer, and messages to be sent to the far-end DR device. The DR32 is simply the conveyer of these messages; it does not examine or add to their content. The controlling process builds command packets and manipulates the three queues, using the four VAX self-relative queue instructions. Figure 4–3 shows the DR32 queue flow. Figure 4–4 shows the contents of a DR32 command packet.

Figure 4–3 DR32 Command Packet Queue Flow

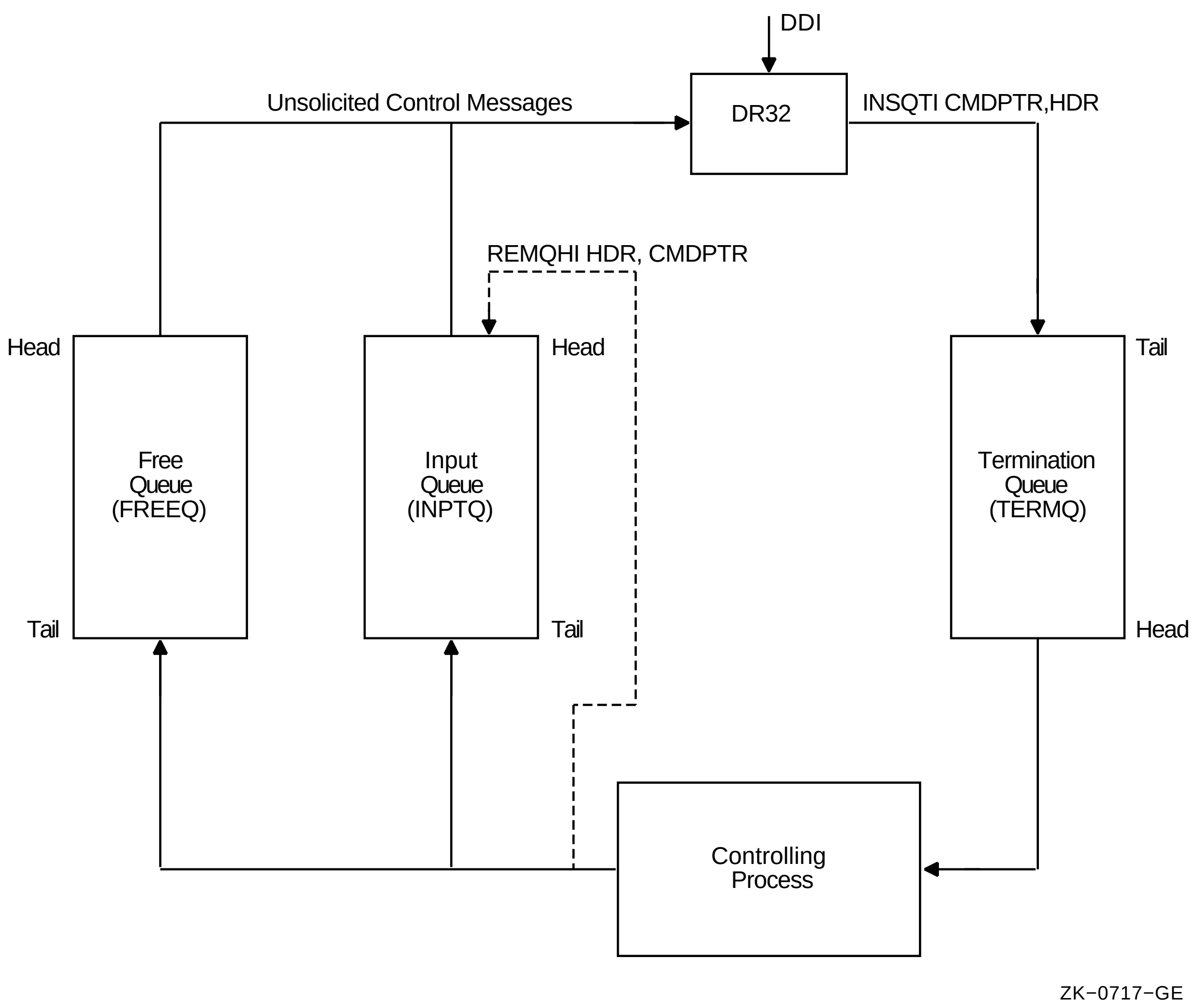
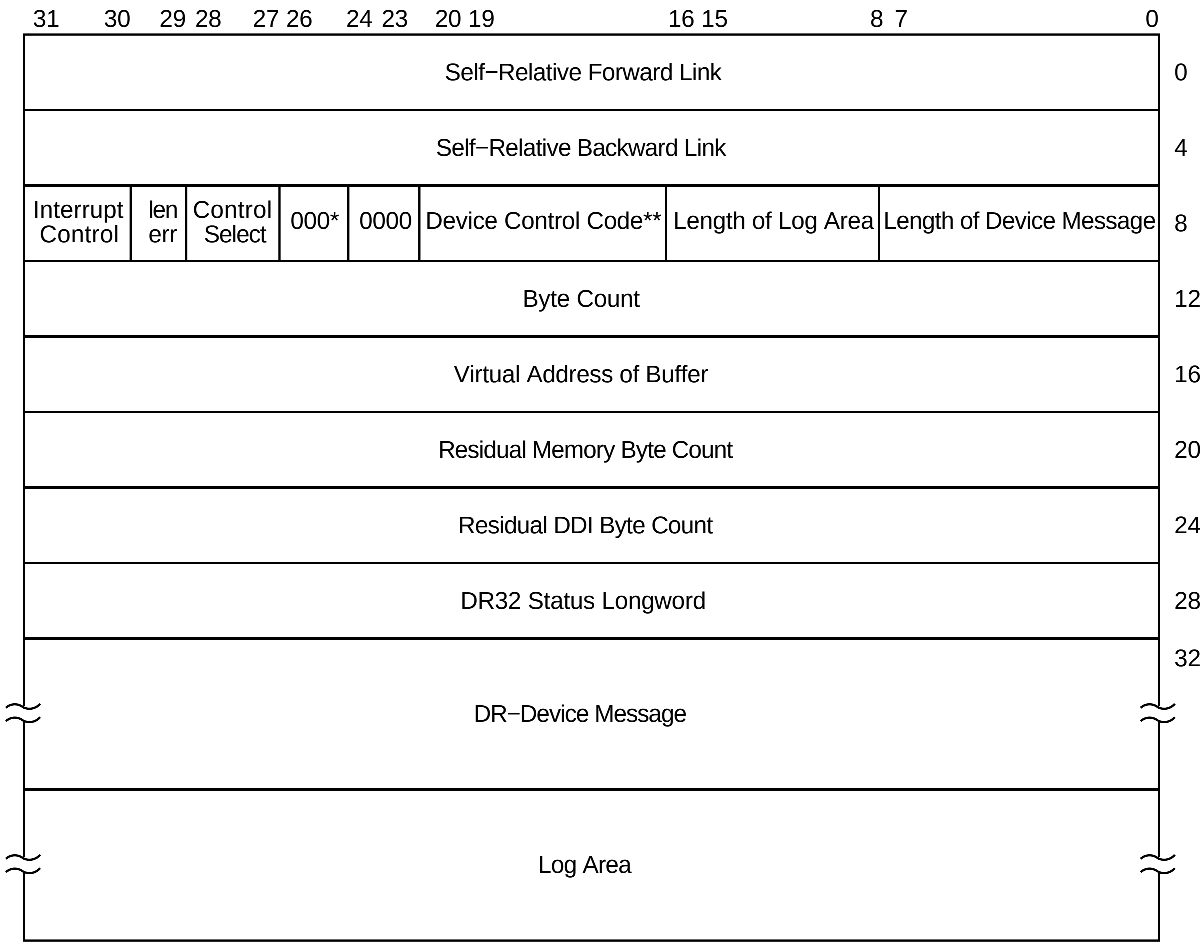


Figure 4–4 DR32 Command Packet



* Bits 31:24 = Packet Control Byte
** Bits 23:16 = Command Control Byte

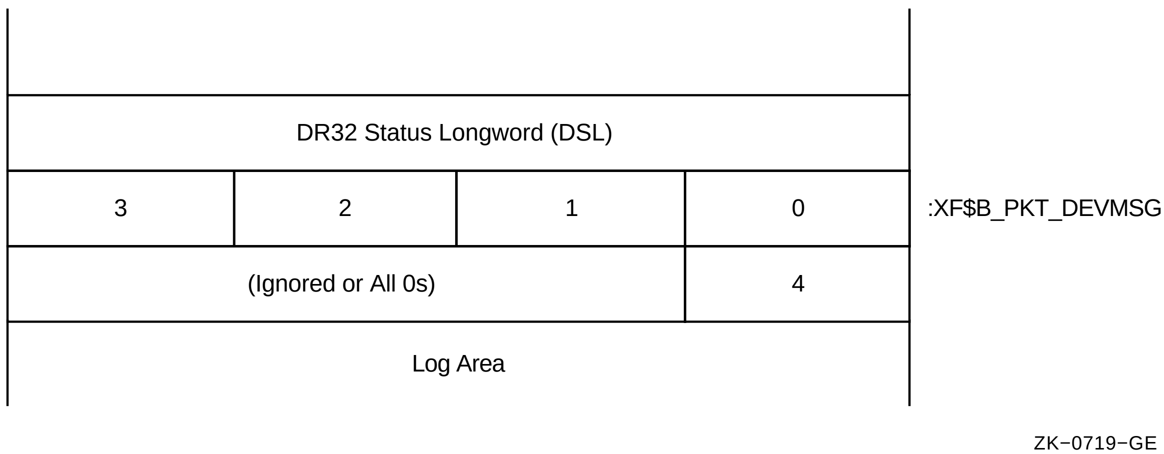
ZK−0718−GE

The following sections provide more information about the command packet fields.

4.4.3.1 Length of Device Message Field

The length of device message field describes the length of the DR device message in bytes. The message length must be less than 256 bytes. Note, however, that the length of device message field itself must always be an integral number of quadwords long. As shown in Figure 4–5, if the application program requires a five-byte device message, it must write a 5 in the length of device message field but allocate eight bytes for the device message field itself. In this case, the last three bytes of the field are ignored by the DR32 when transmitting a message or written as zeros when receiving a message.

Figure 4–5 Detail of the Device Message Field in the Command Packet



The symbolic offset for the length of device message field is XF$B_PKT_MSGLEN.

4.4.3.2 Length of Log Area Field

The length of log area field describes the length of the log area in bytes. The length specified must be less than 256 bytes. Note, however, that the length of log area field itself must be an integral number of quadwords long. For example, if the application program requires a five-byte log area field, it must write a 5 in the length of log area field but allocate eight bytes for the log area field itself. In this case, the last three bytes of the field are written as zeros when receiving a log message (log messages are always received). The symbolic offset for the length of log area field is XF$B_PKT_LOGLEN.

4.4.3.3 Device Control Code Field

The device control field describes the function performed by the DR32. The field occupies the lower half of the command control byte (bits 16 through 23). The operating system defines the following values:

Symbol	Value	Function
XF$K_PKT_RD	0	Read device
XF$K_PKT_RDCHN	1	Read device chained
XF$K_PKT_WRT	2	Write device
XF$K_PKT_WRTCHN	3	Write device chained
XF$K_PKT_WRTCM	4	Write device control message
	5	None; reserved to Digital
XF$K_PKT_SETTST	6	Set self-test
XF$K_PKT_CLRTST	7	Clear self-test
XF$K_PKT_NOP	8	No operation
XF$K_PKT_DIAGRI	9	Diagnostic read internal
XF$K_PKT_DIAGWI	10	Diagnostic write internal
XF$K_PKT_DIAGRD	11	Diagnostic read DDI
XF$K_PKT_DIAGWC	12	Diagnostic write control message

Symbol	Value	Function
XF$K_PKT_SETRND	13	Set random enable
XF$K_PKT_CLRRND	14	Clear random enable
XF$K_PKT_HALT	15	Set halt

Table 4–2 describes the functions performed by the different device control codes.

Table 4–2 Device Control Code Descriptions

Function	Meaning
Read device	Specifies a data transfer from the far-end DR device to the DR32. The control select field (see Section 4.4.3.4) describes the information to be transferred prior to the initiation of the data transfer.
Read device chained	Specifies a data transfer from the far-end DR device to the DR32. The DR32 chains data to the buffer specified in the next command packet in INPTQ. A command packet that specifies the read device chained function must be followed by a command packet that specifies either the read device chained function or the read device function. All other device control codes cause an abort. If a read device chained function is specified, the chain continues. However, if a read device function is specified, that command packet is the last packet in the chain.
Write device and write device chained	Specify data transfers from the DR32 to the far-end DR device. Otherwise, they are similar to read device and read device chained functions.
Write device control message	Specifies the transfer of a control message to the far-end DR device. This message is contained in the device message field of this command packet. The write device control message function directs the controlling DR32 to ignore the byte count and virtual address fields in this command packet.
Set self-test	Directs the DR32 to set an internal self-test flag and to set a disable signal on the DDI. This signal informs the far-end DR device that the DR32 is in self-test mode. While in self-test mode, the DR32 can no longer communicate with the far-end DR device.
Clear self-test	Directs the DR32 to clear the internal self-test flag set by the set self-test function and to return to the normal mode of operation.
No operation	This function explicitly does nothing.

(continued on next page)

Table 4–2 (Cont.) Device Control Code Descriptions

Function	Meaning
Diagnostic read internal	Directs the DR32 to fill the memory buffer, which is described by the virtual address and byte count specified in the current command packet, with the data that is stored in the DR32 data silo. The buffer is filled in a cyclical manner. For example, on the DR780 every 128-byte section of the buffer receives the silo data. The amount of data stored in the buffer equals the DDI byte count minus the SBI byte count. The DDI byte count is equal to the original byte count. No data transmission takes place on the DDI for this function. On the DR780, the diagnostic read internal function destroys the first four bytes in the silo before storing the data in the buffer.
Diagnostic write internal	Together with the diagnostic read internal function, used to test the DR32 read and write capability. The diagnostic write internal function directs the DR32 to store data, which is contained in the memory buffer described by the current command packet, in the DR32 data silo, a FIFO-type buffer. No data transmission takes place on the DDI for this function. The diagnostic write internal function terminates when either of the following conditions occurs: • The memory buffer is empty (the SBI byte count is 0). • An abort has occurred. When the function terminates, the amount of data in the silo equals the DDI byte count minus the SBI memory byte count. (Sections 4.4.3.9 and 4.4.3.10 describe these values.)
Diagnostic read DDI	Tests transmissions over the data portion of the DDI. The DR32 must be in the self-test mode or an abort occurs. On the DR780, the diagnostic read DDI function transmits the contents of DR32 data silo locations 0 to 127 over the DDI and returns the data to the same locations. If data transmission is normal (without errors), the residual memory count is equal to the original byte count, the residual DDI count is 0, and the contents of the silo remain unchanged.
Diagnostic write control message	Tests transmissions over the control portion of the DDI. The DR32 must be in self-test mode or an abort occurs. The diagnostic write control message function directs the DR32 to remove the command packet on FREEQ and check the length of message field. Then the first byte of the message in the command packet on INPTQ is transmitted and read back on the control portion of the DDI. This byte is then written into the message space of the packet from FREEQ. The updated packet from FREEQ is inserted onto TERMQ and is followed by the packet from INPTQ.

(continued on next page)

Table 4–2 (Cont.) Device Control Code Descriptions

Function	Meaning
Set random enable and clear random enable	Directs the DR32 to accept read and write commands sent by the far-end DR device. Range-checking is performed to verify that all addresses specified by the far-end DR device for access are within the buffer block. Far-end DR device-initiated transfers to or from the VAX memory are conducted without notification of the VAX processor or the application program. The clear random enable function directs the DR32 to reject far-end DR device-initiated transfers. Random-access mode must be enabled when the DR32 is used in a processor-to-processor link.
Set halt	Places the DR32 in a halt state. The set halt function always generates a packet interrupt regardless of the value in the interrupt control field (see Section 4.4.3.6). If an AST routine was requested on completion of the QIO function (see Sections 4.4.5.2 and 4.4.6.2), the routine is called after the command packet containing the set halt function has been processed by the DR32.

The following symbolic offsets are defined for the device control code field:

Symbol	Meaning
XF$B_PKT_CMDCTL	Byte offset from the beginning of the command packet
XF$V_PKT_FUNC	Bit offset from XF$B_PKT_CMDCTL
XF$S_PKT_FUNC	Size of the device control code bit field

4.4.3.4 Control Select Field

This field describes the part of the command packet that will be transmitted to the far-end DR device. The control select field is examined only for the read device, read device chained, write device, and write device chained functions; for all others, it is ignored. The operating system defines the following values:

Symbol	Value	Function
XF$K_PKT_NOTRAN	0	No transmission. Nothing is transmitted over the control portion of the DDI. However, if the command packet specifies a data transfer, data can be transmitted over the data portion of the DDI. The primary use of this code is during data chaining.
XF$K_PKT_CB	1	Command control byte (bits 23:16) only. This code directs the DR32 to transmit the contents of the command control byte, which includes the device control code field, to the far-end DR device. This code is used primarily at the start of data chain or nondata chain commands.
XF$K_PKT_CBDM	2	Command control byte and device message. This code directs the DR32 to transmit the command control byte and then the device message. It is used primarily when an interface requires more than one byte of command.

Symbol	Value	Function
XF$K_PKT_CBDMBC	3	Command control byte, device message, and byte count. This code directs the DR32 to transmit the command control byte, the byte count, and the device message (in that order). It is used primarily during processor-to-processor link operations. In this case the device message must be exactly four bytes in length and contain the virtual address of the buffer in the far-end processor's memory.

The following symbolic offsets are defined for the control select field:

Symbol	Meaning
XF$B_PKT_PKTCTL	Byte offset from the beginning of the command packet
XF$V_PKT_CISEL	Bit offset from XF$B_PKT_PKTCTL
XF$S_PKT_CISEL	Size of control select bit field

4.4.3.5 Suppress Length Error Field

The suppress length error field function prevents the DR32 from aborting if the data transfer on the DDI is terminated by the far-end DR device before the DDI byte counter has reached zero.

The following symbolic offsets are defined for the suppress length error field:

Symbol	Meaning
XF$B_PKT_PKTCTL	Byte offset from the beginning of the command packet
XF$V_PKT_SLNERR	Bit offset from XF$B_PKT_PKTCTL
XF$S_PKT_SLNERR	Size of the suppress length error bit field

4.4.3.6 Interrupt Control Field

The interrupt control field determines the conditions under which an interrupt is generated, on a packet-by-packet basis, when the DR32 places this command packet onto TERMQ. Depending on the conditions specified in the IO$_STARTDATA call, the interrupt can set an event flag or call an AST routine.

Symbol	Value	Function
XF$K_PKT_UNCOND	0	Interrupt unconditionally
XF$K_PKT_TMQMT	1	Interrupt only if TERMQ was previously empty
XF$K_PKT_NOINT	2, 3	No interrupt

If the set halt function is active, the interrupt control field is ignored. The set halt function unconditionally causes a packet interrupt. The following symbolic offsets are defined for the interrupt control field:

Symbol	Meaning
XF$B_PKT_PKTCTL	Byte offset from the beginning of the command packet
XF$V_PKT_INTCTL	Bit offset from XF$B_PKT_PKTCTL
XF$S_PKT_INTCTL	Size of the interrupt control bit field

4.4.3.7 Byte Count Field

The byte count field specifies the size in bytes of the data buffer for this data transfer. Together with the virtual address of buffer field, this field describes the buffer in the buffer block that the DR32 will read from or write to.

The following symbolic offset is defined for the byte count field:

Symbol	Meaning
XF$B_PKT_BFRSIZ	Byte offset from the beginning of the command packet

4.4.3.8 Virtual Address of Buffer Field

The virtual address of buffer field specifies the virtual address of the data buffer for this data transfer. Together with the byte count field, this field describes the buffer in the buffer block that the DR32 will read from or write to.

The following symbolic offset is defined for the virtual address of buffer field:

Symbol	Meaning
XF$B_PKT_BFRADR	Byte offset from the beginning of the command packet

4.4.3.9 Residual Memory Byte Count Field

After completion of a read device, read device chained, write device, write device chained, diagnostic read internal, diagnostic write internal, or diagnostic read DDI function specified in this command packet, the DR32 places the packet onto TERMQ for return to the controlling process. At that time, the residual memory byte count field will contain a byte count. The difference between the count specified in the byte count field and the count in this field is the number of bytes transferred to or from physical memory, depending on the direction of transfer.

The following symbolic offset is defined for the residual memory byte count field:

Symbol	Meaning
XF$L_PKT_RMBCNT	Byte offset from the beginning of the command packet

(See also the descriptions of the diagnostic read internal and diagnostic write internal functions in Table 4–2.)

4.4.3.10 Residual DDI Byte Count Field

After completion of a read device, read device chained, write device, write device chained, diagnostic read internal, diagnostic write internal, or diagnostic read DDI function specified in this command packet, the DR32 places the packet onto TERMQ for return to the controlling process. At this time, the residual DDI byte count field contains a byte count. The difference between the count specified in the byte count field and the count in this field is the number of bytes transferred to or from the far-end DR device over the DDI, depending on the direction of transfer.

The following symbolic offset is defined for the residual DDI byte count field:

Symbol	Meaning
XF$L_PKT_RDBCNT	Byte offset from the beginning of the command packet

(See also the descriptions of the diagnostic read internal and diagnostic write internal functions in Table 4–2.)

4.4.3.11 DR32 Status Longword (DSL)

The DR32 stores the final status for a command packet in the DR32 status longword before inserting the packet onto TERMQ. The longword contains two distinct status fields:

31 ... 24	23 ... 16	15 ... 0
0	DDI Status	16 Bits of Status

ZK−0720−GE

Table 4–3 lists the names for the status bits returned in the DR32 status longword.

Table 4–3 DR32 Status Longword (DSL) Status Bits

Name	Meaning
	16 Status Bits
XF$V_PKT_SUCCESS XF$M_PKT_SUCCESS	If set, the command was performed successfully. If not set, one of the following bits must be set: XF$M_PKT_INVPTE XF$M_PKT_RNGERR XF$M_PKT_UNGERR XF$M_PKT_INVPKT XF$M_PKT_FREQMT XF$M_PKT_DDIDIS XF$M_PKT_INVDDI XF$M_PKT_LENERR XF$M_PKT_DRVABT XF$M_PKT_PARERR XF$M_PKT_DDIERR
XF$V_PKT_CMDSTD XF$M_PKT_CMDSTD	If set, the command specified in this packet was started.
XF$V_PKT_INVPTE XF$M_PKT_INVPTE	If set, the DR32 accessed an invalid page table entry.
XF$V_PKT_FREQPK XF$M_PKT_FREQPK	If set, this command packet was removed from FREEQ.
XF$V_PKT_DDIDIS XF$M_PKT_DDIDIS	If set, the far-end DR device is disabled.
XF$V_PKT_SLFTST XF$M_PKT_SLFTST	If set, the DR32 is in self-test mode.
XF$V_PKT_RNGERR XF$M_PKT_RNGERR	If set, a range error occurred; that is, a user-provided address was outside the command block or buffer block.

(continued on next page)

Table 4–3 (Cont.) DR32 Status Longword (DSL) Status Bits

Name	Meaning
	16 Status Bits
XF$V_PKT_UNQERR XF$M_PKT_UNQERR	If set, a queue element was not aligned on a quadword boundary.
XF$V_PKT_INVPKT XF$M_PKT_INVPKT	If set, this packet was not a valid DR32 command packet.
XF$V_PKT_FREQMT XF$M_PKT_FREQMT	If set, a message was received from the far-end DR device and FREEQ was empty.
XF$V_PKT_RNDENB XF$M_PKT_RNDENB	If set, random-access mode is enabled.
XF$V_PKT_INVDDI XF$M_PKT_INVDDI	If set, a protocol error occurred on the DDI.
XF$V_PKT_LENERR XF$M_PKT_LENERR	If set, the far-end DR device terminated the data transfer before the required number of bytes was sent; or a message was received from the far-end DR device, and the device message field in the command packet at the head of FREEQ was not large enough to hold it.
XF$V_PKT_DRVABT XF$M_PKT_DRVABT	The I/O driver aborted the transfer. Usually the result of a Cancel I/O on Channel ($CANCEL) system service request.
XF$V_PKT_PARERR XF$M_PKT_PARERR	A parity error occurred on the data or control portion of the DDI.
	DDI Status
XF$V_PKT_DDISTS XF$S_PKT_DDISTS	DDI status. This field is the one-byte DDI register 0 of the far-end DR device. The following three bits are offsets to this field.
XF$V_PKT_NEXREG XF$M_PKT_NEXREG	An attempt was made to access a nonexistent register in the far-end DR device.
XF$V_PKT_LOG XF$M_PKT_LOG	The far-end DR device registers are stored in the log area.
XF$V_PKT_DDIERR XF$M_PKT_DDIERR	An error occurred on the far-end DR device.

4.4.3.12 Device Message Field

The device message field contains control information to be sent to the far-end DR device. It is used when more than one byte of command is required. The number of bytes in the device message is specified in the length of device message field (see Section 4.4.3.1). (The number of bytes allocated for the length of device message field must be rounded up to an integral number of quadwords.)

If the far-end DR device is a DR32 connected to another processor, a device message can be sent only if the function specified in the device control code field of this command packet is a read device, read device chained, write device, write device chained, or write device control message.

In the case of a write device control message, the data in the device message field is treated as unsolicited input and is written into the device message field of a command packet taken from the far-end DR32's FREEQ.

In the case of a read or write (either chained or unchained) function, the only message allowed is the address of the buffer in the far-end processor that either contains or will receive the data to be transferred. This device message must be exactly four bytes in length. In this case the device message is not stored in the command packet from the far-end DR32's FREEQ, but is used by the far-end DR32 to perform the data transfer.

The device message field is also used in command packets placed on FREEQ to convey unsolicited control messages from the far-end DR device.

The symbolic offset for the device message field is XF$B_PKT_DEVMSG.

4.4.3.13 Log Area Field

The log area field receives the return status and other information from the far-end DR device's DDI registers. Logging must be initiated by the far-end DR device. The presence of a log area does not automatically cause logging to occur.

If the DR32 is connected in a processor-to-processor configuration, the log area field is not used.

4.4.4 DR32 Microcode Loader

The DR32 microcode loader program XFLOADER must be executed prior to using the DR32. Running XFLOADER requires CMKRNL and LOG_IO privileges. Typically, a command to run XFLOADER is placed in the site-specific system startup file. XFLOADER locates the file containing the DR32 microcode in the following manner:

1. XFLOADER attempts to open a file using the logical name XF*c*$WCS, where *c* is the DR32 controller designator. For example, to load microcode on device XFA0, XFLOADER attempts to open a file with the logical name XFA$WCS.
2. If the opening procedure described in step 1 fails, XFLOADER attempts to open the file SYS$SYSTEM:XF780.ULD for a DR780, or SYS$SYSTEM:XF750.ULD for a DR750. This file specification describes the default location and file name for the DR32 microcode.

By default, XFLOADER attempts to load microcode into all DR32s on a system. To limit microcode loading to a subset of DR32s, define the logical name XF$DEVNAM using the device names of the DR32s as the equivalence names. XFLOADER searches for the translation using the LNM$FILE_DEV search list. For example, the following command tells XFLOADER to load microcode only in the first and third DR32s on the system:

```
$ DEFINE/SYSTEM XF$DEVNAM XFA0,XFC0
```

After loading microcode into all specified DR32s, XFLOADER either exits or hibernates, according to the following:

- If XFLOADER was run with an ordinary RUN command (that is, RUN XFLOADER), it exits after loading microcode.
- If XFLOADER was run as a separate process, as with the following command, it hibernates after loading microcode:

  ```
  RUN/UIC=[1,1]/PROCESS=XFLOADER SYS$SYSTEM:XFLOADER
  ```

  In this case, XFLOADER automatically reloads microcode into the DR32s after a power recovery.

XFLOADER performs a load microcode QIO to the DR32 driver.

4.4.5 DR32 Function Codes

The DR32 I/O functions are load microcode and start data transfer. Normally, the controlling process stops data transfers with a set halt command packet. However, the Cancel I/O on Channel ($CANCEL) system service can be used to abort data transfers and complete the I/O operation.

4.4.5.1 Load Microcode

The load microcode function resets the DR32 and loads an image of DR32 microcode. It also sets the DR32 data rate to the last specified value. Physical I/O privilege is required. The operating system defines the following function code:

- 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 DR32
- P2—The number of bytes to be loaded (maximum of 5120 for the DR780)

If any data transfer requests are active when a load microcode request is issued, the load request is rejected and SS$_DEVACTIVE is returned in the I/O status block.

The microcode is verified by addressing each microword and checking for a parity error. (The microcode is not compared to the buffer image.) If there are no parity errors, the microcode was loaded successfully and the driver sets the microcode valid bit in one of the DR32 registers. If there is a parity error, SS$_PARITY is returned in the I/O status block. (The valid bit is cleared by the reset operation.)

In addition to SS$_PARITY, three other status codes can be returned in the I/O status block: SS$_NORMAL, SS$_DEVACTIVE, and SS$_POWERFAIL.

4.4.5.2 Start Data Transfer

The start data transfer function specifies a command table that holds the parameters required to start the DR32. In addition to several other parameters, the command table contains the size and address of the command and buffer blocks and the address of a command packet AST routine. No user privilege is required. The operating system defines the following function code:

- IO$_STARTDATA—Start data transfer

The start data transfer function takes the following function modifier:

- IO$M_SETEVF—Set event flag

If IO$M_SETEVF is included with the function code, the specified event flag is set when a command packet interrupt occurs and when the start data transfer QIO is completed. If IO$M_SETEVF is not specified, the event flag is set only when the QIO is completed.

IO$M_SETEVF should not be used with the $QIOW macro because $QIOW will return after the event flag is set the first time.

The start data transfer 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 32) of the data transfer command table (the symbolic name is XF$K_CMT_LENGTH)

The format of the data transfer command table is shown in Figure 4–6 (offsets are shown in parentheses).

Figure 4–6 Data Transfer Command Table

			Offset
Command Block Size (XF$L_CMT_CBLKSZ)			0
Command Block Address (XF$L_CMT_CBLKAD)			4
Buffer Block Size (XF$L_CMT_BBLKSIZ)			8
Buffer Block Address (XF$L_CMT_BBLKAD)			12
Command Packet AST Routine Address (XF$L_CMT_PASTAD)			16
Command Packet AST Parameter (XF$L_CMT_PASTPM)			20
	Flags (XF$B_CMT_FLAGS)	Data Rate (XF$B_CMT_RATE)	24
Address of the Location to Store the GO Bit Address (XF$L_CMT_GBITAD)			28

ZK−0721−GE

Because the command block contains the queue headers for INPTQ, TERMQ, and FREEQ, its address in the second longword must be quadword-aligned.

The command packet AST routine specified in the fifth longword is called whenever the DR32 signals a command packet interrupt. A command packet AST should be distinguished from a QIO AST (**astadrs** argument). A command packet interrupt occurs whenever the DR32 completes a function and returns a packet that specifies an interrupt (see Section 4.4.3.6) by inserting it onto TERMQ. The **astadrs** argument address is called when the QIO is completed. If either the command packet AST address or the **astadrs** address is zero, the respective AST is not delivered. If the command packet specifies the set halt function, a command packet interrupt occurs regardless of the state of the packet interrupt bits.

The seventh longword contains the data rate byte and a flags byte. The data rate byte controls the DR32 clock rate. The data rate value is considered to be an unsigned integer.

For the DR780, the relationship between the value of the data rate byte and the actual data rate is given by the following formula:

$$Data\ rate\ (in\ megabytes/sec) = \frac{40}{256 - (value\ of\ data\ rate\ byte)}$$

For example, a data rate value of 236 corresponds to an actual data rate of 2.0 megabytes/second.

For the DR750, use the following formula:

$$Data\ rate\ (in\ megabytes/sec) = \frac{12.50}{256 - (value\ of\ data\ rate\ byte)}$$

For example, a data rate value of 236 corresponds to an actual data rate of 0.625 megabytes/second.

The maximum data rate byte values are 250 megabytes/second for the DR780 and 252 megabytes/second for the DR750.

The parameter XFMAXRATE set during system generation limits the maximum data rate that can be set. This parameter limits the maximum data rate because very high data rates on certain configurations can cause a processor timeout. If you attempt to set the data rate higher than the rate allowed by XFMAXRATE, the error status SS$_BADPARAM is returned in the I/O status block.

The operating system defines the following flag bit values:

XF$V_CMT_SETRTE	If set, XF$B_CMT_RATE specifies the data rate. If clear, the data rate established by a previous $IO_STARTDATA function is used. The IO$_LOADMCODE function sets the data rate to the last value used. If the data rate has not been previously set, a value of 0 is used.
XF$V_CMT_DIPEAB	If set, parity errors on the data portion of the DDI do not cause device aborts. If clear, a parity error results in a device abort.

The eighth longword contains the address of a location to store the address of the GO bit. This bit must be set whenever the application program inserts a command packet onto an empty INPTQ. The GO bit register is mapped in system memory space and the address is returned to the user.

The IO$_STARTDATA function locks the command and buffer blocks into memory and starts the DR32. Whenever the DR32 interrupts with a command packet interrupt, the driver queues a packet AST (if an AST address is specified) and, if IO$M_SETEVF is specified, sets the event flag. The QIO remains active until one of the following events occurs:

- A set halt command packet is processed by the DR32.
- The data transfer aborts.
- A Cancel I/O on Channel ($CANCEL) system service is issued on this channel.

If an abort occurs, the second longword of the I/O status block contains additional bits that identify the cause of the abort (see Section 4.5).

The start data transfer function can return the following error codes in the I/O status block:

SS$_ABORT	SS$_BUFNOTALIGN	SS$_CANCEL
SS$_CTRLERR	SS$_DEVREQERR	SS$_EXQUOTA
SS$_INSFMEM	SS$_IVBUFLEN	SS$_MCNOTVALID
SS$_NORMAL	SS$_PARITY	SS$_POWERFAIL

4.4.6 High-Level Language Interface

The OpenVMS VAX operating system supports a set of program-callable procedures that provide access to the DR32. The formats of these calls are given here for VAX FORTRAN users; VAX MACRO users must set up a standard OpenVMS argument block and issue the standard procedure CALL. (Optionally, VAX MACRO users can access the DR32 directly by issuing an IO$_STARTDATA function, building command packets, and inserting them onto INPTQ.) Users of other high-level languages can also specify the proper subroutine or procedure invocation.

Six high-level language procedures are provided by the OpenVMS VAX operating system for the DR32. They are contained in the default system library, STARLET.OLB. Table 4–4 lists these procedures. Procedure arguments are either input or output arguments, that is, arguments supplied by the user or arguments that will contain information stored by the procedure. Except for those that are indicated as output arguments, all arguments in the following call descriptions are input arguments. By default, all procedure arguments are integer variables unless otherwise indicated.

The OpenVMS high-level language support routines for the DR32 do the following:

- Issue I/O requests
- Allocate and manage the command memory
- Build command packets, insert them onto INPTQ, and set the GO bit
- Remove command packets from TERMQ and return the information they contain to the controlling process
- Use action routines for program–DR32 synchronization

Table 4–4 Operating System Procedures for the DR32

Subroutine	Function
XF$SETUP	Defines command and buffer areas and initializes queues
XF$STARTDEV	Issues an I/O request that starts the DR32
XF$FREESET	Releases command packets onto FREEQ
XF$PKTBLD	Builds command packets and releases them onto INPTQ
XF$GETPKT	Removes a command packet from TERMQ
XF$CLEANUP	Deassigns the device channel and deallocates the command area

The operating system also provides a FORTRAN parameter file, SYS$LIBRARY:XFDEF.FOR, that can be included in FORTRAN programs. This file defines many of (but not all) the symbolic names with the XF$ prefix described in this chapter. For example, SYS$LIBRARY:XFDEF.FOR contains symbolic definitions for function codes (that is, device control codes), interrupt control codes, command control codes, and masks for error bits set in the I/O status block and the DR32 status longword. To include these definitions in a FORTRAN program, insert the following statement in the source code:

```
INCLUDE 'SYS$LIBRARY:XFDEF.FOR'
```

4.4.6.1 XF$SETUP

The XF$SETUP subroutine defines memory space for the command and buffer areas and initializes INPTQ, TERMQ, and FREEQ. The call to XF$SETUP must be made prior to any calls to other DR32 support routines.

The format of the XF$SETUP call is as follows:

```
CALL XF$SETUP(contxt,barray,bufsiz,numbuf,[idevmsg],[idevsiz],-
              [ilogmsg],[ilogsiz],[cmdsiz],[status])
```

Argument descriptions are as follows:

contxt A 30-longword user-supplied array that is maintained by the support routines and is used to contain context and status information concerning the current data transfer (see Section 4.4.6.5). The **contxt** array provides a common storage area that all support routines share. For increased performance, **contxt** should be longword-aligned.

barray Specifies the starting virtual address of an array of buffers that, in the case of an output operation, contain information for transfer by the DR32, or in the case of an input operation, will contain information transferred by the DR32. For example, if **barray** is declared INTEGER*2 BARRAY (I,J), I is the size of each data buffer in words and J is the number of buffers. The lower bound on both indexes is assumed to be 1. All buffers in the array must be contiguous to each other and of fixed size.

bufsiz Specifies the size in bytes of each buffer in the array. All buffers are the same size. If the **barray** argument is declared as stated in the preceding paragraph, **bufsiz** = I*2. The **bufsiz** argument length is one longword.

numbuf Specifies the number of buffers in the array. If the **barray** argument is declared as in the preceding paragraph, **numbuf** = J. The area of memory described by the **barray**, **bufsiz**, and **numbuf** arguments is used as the buffer block for DR32 data transfers. The **numbuf** argument length is one longword.

idevmsg Specifies an array, declared by the application program, that is used to store an unsolicited input device message from the far-end DR device. The DR32 stores unsolicited input in the device message field of a command packet from FREEQ and places that packet onto TERMQ. When XF$GETPKT removes such a packet from TERMQ, it copies the device message field into the **idevmsg** array. The calling program is then notified that information has been stored in the **idevmsg** array. The **idevmsg** argument is optional; the argument must be given if any unsolicited input is anticipated.

idevsiz Specifies the size in bytes of the **idevmsg** array. The maximum size of a device message is 256 bytes. The **idevsiz** argument is optional; if **idevmsg** is specified, **idevsiz** must be specified. The **idevsiz** argument length is one word.

ilogmsg Specifies an array, declared by the application program, that is used to store log information from the far-end DR device contained in the log area field of the command packet. Log information is hardware-dependent data that is returned by the far-end DR device. The XF$SETUP routine stores the address and size of the **ilogmsg** array; the log information is stored in the **ilogmsg** array by the XF$GETPKT routine. The **ilogmsg** argument is optional; the argument must be given if any log information is anticipated.

ilogsiz Specifies the size in bytes of the **ilogmsg** array. The maximum size of a log message is 256 bytes. The **ilogsiz** argument is optional. However, if **ilogmsg** is specified, **ilogsiz** must be specified. The **ilogsiz** argument length is one word.

cmdsiz Specifies the amount of memory space to be allocated from which command packets are to be built. Consider the following factors when deciding how much memory to allocate for this purpose:

- The number of command packets that the application program will be using.
- The device message and log area fields in command packets are rounded up to quadword boundaries.
- The size of the command packet itself is rounded up to an eight-byte boundary.
- **cmdsiz** will be rounded up to a page boundary.

The **cmdsiz** argument is optional; argument length is one longword. If defaulted, the allocated space is equal to the following, which is rounded up to a full page:

(**numbuf**)*(32+**idevsiz**+**ilogsiz**)*(3)

Memory space for command packets is obtained by calling LIB$GET_VM.

status This output argument receives the operating system success or failure code of the XF$SETUP call:

SS$_NORMAL	Normal successful completion
SS$_BADPARAM	Invalid input argument

Error returns can be found in LIB$GET_VM.

The **status** argument is optional; argument length is one longword.

4.4.6.2 XF$STARTDEV

The XF$STARTDEV subroutine issues the I/O request that starts the DR32 data transfer.

The format of the XF$STARTDEV call is as follows:

CALL XF$STARTDEV(contxt,devnam,[pktast],[astparm],[efn],[modes],[datart],[status])

Argument descriptions are as follows:

contxt Specifies the array that contains context and status information (see Section 4.4.6.1).

devnam Specifies the device name (logical name or actual device name) of the DR32. All letters in the resultant string must be capitalized and the device name must terminate with a colon, for example, XFA0:. The **devnam** data type is character string.

pktast Specifies the address of an AST routine that is called each time a command packet that specifies an interrupt in its interrupt control field is returned by the DR32, that is, placed onto TERMQ (see Section 4.4.7.2). This AST routine is also called on completion of the I/O request. Normally, the AST routine would call XF$GETPKT to remove command packets from TERMQ until TERMQ is empty. The **pktast** argument is optional.

astparm Specifies a longword parameter that is included in the call to the **pktast**-specified AST routine. The format used to call the AST routine is:

CALL pktast(astparm)

The **astparm** argument is optional; argument length is one longword. If **astparm** is not specified, **pktast** is called with no parameter.

efn If the event flag must be determined by the application program, **efn** specifies the number of the event flag that is set when a packet interrupt is delivered. Otherwise, it is not necessary to include this argument in an XF$STARTDEV call. If defaulted, **efn** is 21. The **efn** argument length is one word.

The event flag (either the default or the event flag specified by this argument) is set for every packet interrupt and also when the QIO completes.

modes Specifies the mode of operation. The operating system defines the following value:

2 = parity errors on the data portion of the DDI that do not cause the device to abort.

If defaulted, **modes** is 0 (a parity error causes the device to abort).

datart Specifies the data rate. The data rate controls the speed at which the transfer takes place. The data rate is considered to be an unsigned integer in the range 0 to 255. The relationship between the specified data rate value and the actual data rate for the DR780 and the DR750 is shown in Section 4.4.5.2.

If **datart** is defaulted, the previously set data rate is used. The **datart** argument length is one byte.

status This output argument receives the operating system success or failure code of the XF$STARTDEV call:

SS$_NORMAL	Normal successful completion
SS$_BADPARAM	Required parameter defaulted

Error returns can be obtained by issuing the Create I/O on Channel ($CREATE) and Queue I/O Request ($QIO) system services.

The **status** argument is optional; argument length is one longword.

4.4.6.3 XF$FREESET

The XF$FREESET subroutine releases command packets onto FREEQ. These packets are then available to the DR780 to store any unsolicited input from the far-end DR device. If unsolicited input from the far-end DR device is expected, the XF$FREESET call should be made before the XF$STARTDEV call is issued.

idevsiz, the argument that specifies the size of the **idevmsg** array in the call to XF$SETUP, defines the size of the device message field in command packets inserted onto FREEQ. This occurs because unsolicited device messages are copied from the device message field of the command packet to the **idevmsg** array.

Note that the XF$FREESET subroutine may occasionally disable ASTs for a very short period.

The format of the XF$FREESET call is as follows:

```
CALL XF$FREESET(contxt,[numpkt],[intctrl],[action],[actparm],[status])
```

Argument descriptions are as follows:

contxt Specifies the array that contains context and status information (see Section 4.4.6.1).

numpkt Specifies the number of command packets to be released onto FREEQ. The **numpkt** argument is optional; argument length is one word. If defaulted, **numpkt** is 1.

intctrl Specifies the conditions under which an AST is delivered (and the event flag set) when the DR32 places this command packet (or packets) on TERMQ (see Section 4.4.6.2). The operating system defines the following values:

0 = unconditional AST delivery and event flag set
1 = AST delivery and event flag set only if TERMQ is empty
2 = no AST interrupt or event flag set

The **intctrl** argument is optional; argument length is one word. If defaulted, **intctrl** is 0.

action Specifies the address of a routine that is called when any command packet built by this call to XF$FREESET is removed from TERMQ by XF$GETPKT (see Section 4.4.7.3). The **action** argument is optional.

actparm A longword parameter that is passed to the action routine when the action routine is called (see Section 4.4.7.3). The **actparm** argument is optional.

status This output argument receives the operating system success or failure code of the XF$FREESET call:

SS$_NORMAL	Normal successful completion
SS$_BADQUEUEHDR	FREEQ interlock timeout
SS$_INSFMEM	Insufficient memory to build command packets
SHR$_NOCMDMEM	Command memory not allocated (usually because the data transfer has stopped and XF$CLEANUP has been called or because XF$SETUP has not been called)

4.4.6.4 XF$PKTBLD

The XF$PKTBLD subroutine builds command packets and releases them onto INPTQ.

Note that the XF$PKTBLD subroutine may occasionally disable ASTs for a very short period.

The format of the XF$PKTBLD call is as follows:

```
CALL XF$PKTBLD(contxt,func,[index],[size],[devmsg],[devsiz],[logsiz],-
               [modes],[action],[actparm],[status])
```

Argument descriptions are as follows:

contxt Specifies the array that contains context and status information (see Section 4.4.6.1).

func Specifies the device control code. Device control codes describe the function the DR32 is to perform. The **func** argument length is one word. Table 4–2 describes the functions in greater detail. The operating system defines the following values:

Symbol	Value	Function
XF$K_PKT_RD	0	Read device
XF$K_PKT_RDCHN	1	Read device chained
XF$K_PKT_WRT	2	Write device
XF$K_PKT_WRTCHN	3	Write device chained
XF$K_PKT_WRTCM	4	Write device control message
	5	None; reserved to Digital
XF$K_PKT_SETTST	6	Set self-test
XF$K_PKT_CLRTST	7	Clear self-test
XF$K_PKT_NOP	8	No operation
XF$K_PKT_DIAGRI	9	Diagnostic read internal
XF$K_PKT_DIAGWI	10	Diagnostic write internal
XF$K_PKT_DIAGRD	11	Diagnostic read DDI
XF$K_PKT_DIAGWC	12	Diagnostic write control message
XF$K_PKT_SETRND	13	Set random enable
XF$K_PKT_CLRRND	14	Clear random enable
XF$K_PKT_HALT	15	Set halt

index
Specifies the index of a data buffer specified by the **barray** argument (see Section 4.4.6.1). The specific index value given means that elements **barray** (1,index) through **barray** (size,index) will be transferred (one buffer full of data). The **index** argument is optional and is used only when the function specifies a data transfer (a read device, read device chained, write device, or write device chained function). The **index** argument length is one word.

size
Specifies a byte count to be transferred. This argument is optional and is used only when the function specifies a data transfer. If defaulted, the number of bytes to be transferred is assumed to be the size of the buffer (specified by the **bufsiz** argument in the call to XF$SETUP). If the **size** argument is given, the specified number of bytes of data **barray** (1,index) through **barray** (size,index) will be transferred. If **size** is defaulted and the function specifies a data transfer, **barray** (1,index) through **barray** (bufsiz,index) will be transferred. The **size** argument length is one longword.

devmsg
Specifies a variable that contains the device message to be sent to the far-end DR device. Provides additional control of the far-end DR device (see Section 4.4.3.12). The **devmsg** argument is optional.

devsiz
Specifies the size in bytes of the **devmsg** variable. If the **modes** argument specifies that a device message is to be sent over the control portion of the DDI, **devsiz** specifies the number of bytes of **devmsg** that will be sent to the far-end DR device.

logsiz
Specifies the size of the log message expected from the far-end DR device. The **logsiz** argument is optional; argument length is one word. If defaulted, **logsiz** is 0.

modes Provides additional control of the transaction. The operating system defines the following values:

Value	Meaning
+8	Only the function code is sent over the control portion of the DDI to the far-end DR device. Only for read device, read device chained, write device, and write device chained functions.
+16	The function code and the device message are sent over the control portion of the DDI to the far-end DR device. Only for read device, read device chained, write device, and write device chained functions.
+24	The function code, the device message, and the buffer size are sent over the control portion of the DDI to the far-end DR device. Only for read device, read device chained, write device, and write device chained functions.
	If none of the preceding three values is selected, nothing is transmitted over the control portion of the DDI to the far-end DR device.
+32	Length errors are suppressed. If not selected, a length error results in an abort.
+64	An AST should be delivered (and an event flag set) when this command packet is inserted onto TERMQ, provided TERMQ is empty.
+128	No AST is delivered or event flag set for this command packet.
	If both +64 and +128 are selected, +128 takes precedence.
	If neither of the preceding two values is selected, ASTs are delivered and the event flag is set unconditionally (whenever this command packet is placed onto TERMQ).
+256	Insert this command packet at the head of INPTQ. If not selected, insert the packet at the tail of INPTQ.

The **modes** argument default value is 0.

action Specifies the address of a routine that is called when XF$GETPKT removes this command packet from TERMQ. This occurs after the DR32 has completed the command specified in the packet (see Section 4.4.7.3). The **action** argument length is one longword.

actparm A longword parameter that is passed to the action routine when the action routine is called (see Section 4.4.7.3). The **actparm** argument is optional.

status This output argument receives the operating system success or failure code of the XF$PKTBLD call:

SS$_NORMAL	Normal successful completion
SS$_BADPARAM	Input parameter error
SS$_BADQUEUEHDR	INPTQ interlock timeout
SS$_INSFMEM	Insufficient memory to build command packets
SHR$_NOCMDMEM	Command memory not allocated (usually because the data transfer has stopped and XF$CLEANUP has been called or because XF$SETUP has not been called)

4.4.6.5 XF$GETPKT

The XF$GETPKT subroutine removes a command packet from TERMQ.

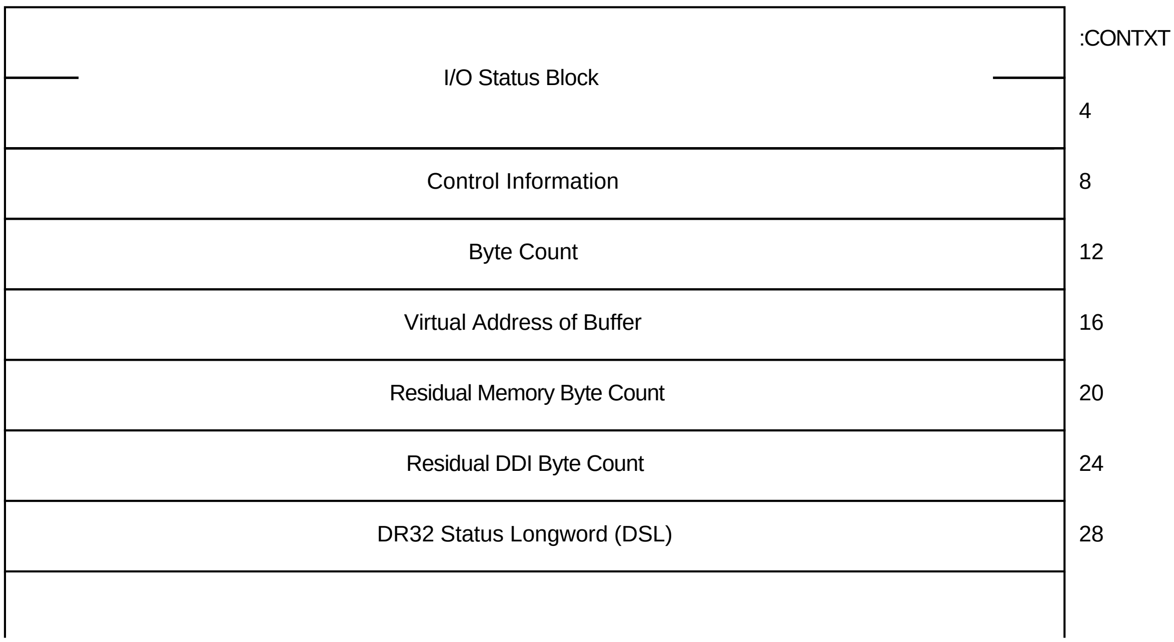
Note that the XF$GETPKT subroutine may occasionally disable ASTs for a very short period.

The format of the XF$GETPKT call is as follows:

CALL XF$GETPKT(contxt,[waitflg],[func],[index],[devflag],[logflag],[status])

Argument descriptions are as follows:

contxt Specifies the array that contains the context and status information (see Section 4.4.6.1). On return from XF$GETPKT, the first eight longwords of the **contxt** array are filled with the status of the data transfer:



ZK−0722−GE

The first two longwords are the I/O status block. The next six longwords are copied directly from bytes 8 through 31 of the command packet.

This context and status information is returned by the DR32 as status in each command packet. With the exception of the I/O status block, the information is copied by XF$GETPKT into the **contxt** array whenever XF$GETPKT removes a command packet from TERMQ.

The I/O status block is stored only after the data transfer has halted and it contains the final status of the transfer. Section 4.5 describes the I/O status block.

(See Section 4.4.2 for a description of the remaining fields.)

waitflg Specifies the consequences of an attempt by XF$GETPKT to remove a command packet from an empty TERMQ. If **waitflg** is 0 (default), XF$GETPKT waits for the event flag to be set and then removes a packet from TERMQ. If **waitflg** is 1, XF$GETPKT returns immediately with a failure status. Normally, **waitflg** is set to 1 (.TRUE.) for AST synchronization and set to 0 (.FALSE.) for event flag synchronization (see Section 4.4.7). The **waitflg** argument is optional.

func This output argument receives the device control code specified in this command packet (see Section 4.4.6.4). The **func** argument is optional; argument length is one word.

index If the current command packet specified a data transfer, this output argument receives the buffer index specified when this command packet was built by XF$PKTBLD (see Section 4.4.6.4). The **index** argument is optional; argument length is one word.

devflag If set to .TRUE. (255), this output argument indicates that a device message was stored in the **idevmsg** array, which is described in the XF$SETUP call (see Section 4.4.6.1). The **devflag** argument is optional; argument length is one byte.

logflag If set to .TRUE. (255), this output argument indicates that a log message was stored in the **ilogmsg** array, which is described in the XF$SETUP call (see Section 4.4.6.1). The **logflag** argument is optional; argument length is one byte.

status This output argument receives the status of the XF$GETPKT call:

SS$_NORMAL	Normal successful completion
SS$_BADQUEUEHDR	TERMQ interlock timeout
SHR$_QEMPTY	TERMQ empty but transfer still in progress; only returned if **waitflg** is .TRUE.
SHR$_HALTED	TERMQ empty, transfer complete, and I/O status block contains final status; XF$CLEANUP called automatically (subsequent calls to XF$GETPKT return SHR$_NOCMDMEM)
SHR$_NOCMDMEM	Command memory not allocated; usually indicates either: **1** XF$SETUP not called **2** XF$CLEANUP called

4.4.6.6 XF$CLEANUP

The XF$CLEANUP subroutine deassigns the channel and deallocates the command area allocated by XF$SETUP. If XF$GETPKT detects a TERMQ empty condition and the transfer has halted, it will automatically call XF$CLEANUP. However, if the transfer either terminates in an SS$_CTRLERR or SS$_BADQUEUEHDR error, or is intentionally terminated, XF$GETPKT might not detect these conditions and XF$CLEANUP should be called explicitly.

The format of the XF$CLEANUP call is as follows:

```
CALL XF$CLEANUP(contxt,[status])
```

Argument descriptions are as follows:

contxt Specifies the array that contains context and status information (see Section 4.4.6.1).

status This output argument receives the status of the XF$CLEANUP call:

SS$_NORMAL	Normal successful completion
SHR$_NOCMDMEM	The command memory not allocated; there are error returns from LIB$FREE_VM and $DASSIGN

4.4.7 User Program DR32 Synchronization

Synchronization of high-level language application programs with the DR32 can be achieved in the following ways:

- Event flags
- AST routines
- Action routines

4.4.7.1 Event Flags

Event flags are synchronized by calling the XF$GETPKT routine (see Section 4.4.6.5) with the **waitflg** argument set to 0 (default). The **pktast** argument in the XF$STARTDEV routine (see Section 4.4.6.2) is normally set to its default value. If the XF$GETPKT routine is called and the termination queue is empty, the routine waits until the DR32 places a command packet on the queue and sets the event flag. The packet is then removed from the queue and returned to the caller.

4.4.7.2 AST Routines

If a call to the XF$STARTDEV routine includes the **pktast** argument, the specified AST routine is called each time an AST is delivered. AST delivery can be controlled on a packet-by-packet basis by using the **intctrl** argument in the XF$FREESET routine and by specifying appropriate values in the **modes** argument of the XF$PKTBLD routine (see Sections 4.4.6.3 and 4.4.6.4). For a particular command packet, ASTs can be delivered as follows:

- Unconditionally when the packet is placed onto TERMQ
- Only if TERMQ is empty when the packet is placed on it
- Not at all (that is, there is no AST when the packet is placed on TERMQ)

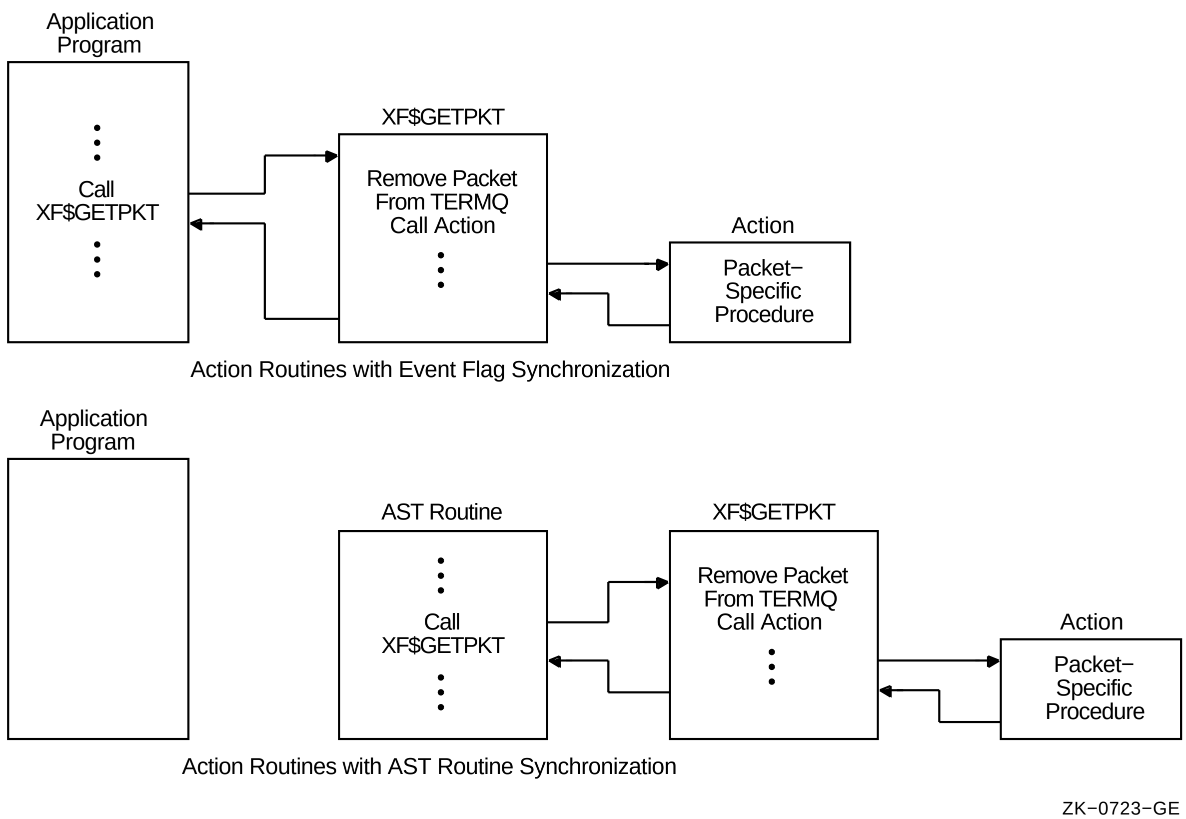
There is no guarantee that an AST will be delivered for every command packet, even when the **astctrl** argument indicates unconditional AST delivery. In particular, if packet interrupts are closely spaced, several packets can be placed onto TERMQ even though only one AST is delivered. Therefore, the AST routine should continue to call the XF$GETPKT routine until all command packets are removed from TERMQ.

4.4.7.3 Action Routines

The **action** argument specified in the XF$FREESET and XF$PKTBLD routines (see Sections 4.4.6.3 and 4.4.6.4) can be used for a more automated synchronization of the program with the DR32. Routines specified by **action** arguments can be used for both event flag and AST routine synchronization.

The address of the action routine is included in the command packet. This routine is automatically called by the XF$GETPKT routine when it removes that packet from TERMQ. This allows you to define, at the time the command packet is built, how it will be handled once it is removed from TERMQ. In addition to specifying different action routines for different types of command packets, you can also specify an action routine parameter (**actparm**) to further identify the command packet or the action to be taken when the command is completed. Figure 4–7 shows the use of action-specified routines for program synchronization.

Figure 4–7 Action Routine Synchronization



An important difference between AST routine and action routine use is the number of times the respective routines are specified. Command packet AST routines are specified only once, in an XF$STARTDEV call; a single AST routine is implied. Action routines, however, are specified in each command packet. This allows a different action routine to be designed for each type of command packet.

Routines specified by the action argument are supplied by the user. The format of the calling interface is as follows:

CALL action-routine (contxt,actparm,devflag,logflag,func,index,status)

With the exception of **actparm**, all arguments are the same as those described for the XF$GETPKT routine. In effect, the action routine receives the same information XF$GETPKT optionally returns to its calling program, along with the **actparm** argument that was specified when the packet was built. If these variables are to be passed as input to the action routine, they must be supplied as output variables in the call to the XF$GETPKT routine.

4.5 I/O Status Block

The I/O status block for the load microcode and start data transfer QIO functions is shown in Figure 4–8. The I/O status block used in the first two longwords of the **contxt** array for high-level language calls also has the same format.

Figure 4–8 IOSB Contents for the DR32 Functions

31–27	26–24	23–16	15–0
0			Status
5 Status Bits	0	DDI Status	16 Status Bits

ZK−0724−GE

The operating system status values are returned in the first longword. Appendix A lists these values. (The OpenVMS system messages documentation provides explanations and user actions for these returns.) If SS$_CTRLERR, SS$_DEVREQERR, or SS$_PARITY is returned in the status word, the second longword contains additional returns (device-dependent data). Table 4–5 lists these returns.

The I/O status block for an I/O function is returned after the function completes. Status is not stored on the completion of every command packet because any number of packets can pass between the application program and the DR32 when a single QIO executes.

Table 4–5 Device-Dependent IOSB Returns for I/O Functions

Symbolic Name	Meaning
	16 Status Bits
XF$V_PKT_SUCCESS	The command was performed successfully.
XF$V_IOS_CMDSTD	The command specified in the command packet started.
XF$V_IOS_INVPTE	An invalid page table entry.
XF$V_IOS_FREQPK	This command packet came from FREEQ.
XF$V_IOS_DDIDIS	The far-end DR device is disabled.
XF$V_IOS_SLFTST	The DR32 is in self-test mode.
XF$V_IOS_RNGERR	The user-provided address is outside the command block range or the buffer block range.
XF$V_IOS_UNQERR	A queue element was not aligned on a quadword boundary.
XF$V_IOS_INVPKT	A packet was not a valid DR32 command packet.
XF$V_IOS_FREQMT	A message was received from the far-end DR device and FREEQ was empty.
XF$V_IOS_RNDENB	Random-access mode is enabled.
XF$V_IOS_INVDDI	A protocol error occurred on the DDI.
XF$V_IOS_LENERR	The far-end DR device terminated the data transfer before the required number of bytes was sent, or a message was received from the far-end DR device and the device message field in the command packet at the head of FREEQ was not large enough to hold it.
XF$V_IOS_DRVABT	The I/O driver aborted the DR32 function.

(continued on next page)

Table 4–5 (Cont.) Device-Dependent IOSB Returns for I/O Functions

Symbolic Name	Meaning
	16 Status Bits
XF$V_PKT_PARERR	A parity error occurred on the data or control portion of the DDI.
	DDI Status
XF$V_IOS_DDISTS	The one-byte status register 0 for the far-end DR device. XFV_IOS_NEXREG, XFV_IOS_LOG, and XF$V_IOS_DDIERR are returns from this register.
XF$V_IOS_NEXREG	An attempt was made to access a nonexistent register on the far-end DR device.
XF$V_IOS_LOG	The far-end DR device registers are stored in the log area.
XF$V_IOS_DDIERR	An error occurred on the far-end DR device.
	5 Status Bits
XF$V_IOS_BUSERR	An error on the processor's internal CPU memory bus occurred.
XF$V_IOS_RDSERR	A noncorrectable memory error occurred (read data substitute).
XF$V_IOS_WCSPE	Writable control store (WCS) parity error.
XF$V_IOS_CIPE	Control interconnect parity error. A parity error occurred on the control portion of the DDI.
XF$V_IOS_DIPE	Data interconnect parity error. A parity error occurred on the data portion of the DDI.

4.6 Programming Hints

This section contains information about important programming considerations relevant to users of the DR32 driver.

4.6.1 Command Packet Prefetch

The DR32 has the capability of prefetching command packets from INPTQ. While executing the command specified in one packet, the DR32 can prefetch the next packet, decode it, and be ready to execute the specified command at the first opportunity. When the command is executed depends on which command is specified. For example, if two read device or write device command packets are on INPTQ, the DR32 fetches the first packet, decodes the command, verifies that the transfer is legal, and starts the data transfer. While the transfer is taking place, the DR32 prefetches the next read device or write device command packet, decodes it, and verifies the transfer legality. The second transfer begins as soon as the first transfer is completed.

If the two command packets on INPTQ are read device (or write device) and write device control message, in that order, the DR32 prefetches the second packet and immediately executes the command, because control messages can be overlapped with data transfers. The DR32 then prefetches the next command packet. In an extreme case, the DR32 can send several control messages over the control portion of the DDI while a single data transfer takes place on the data portion of the DDI.

The prefetch capability and the overlapping of control and data transfers can cause unexpected results when programming the DR32. For instance, if the application program calls for a data transfer to the far-end DR device followed by notification of the far-end DR device that data is present, the program cannot simply insert a write device command packet and then a write control message command packet onto INPTQ—the control message might arrive before the data transfer completes.

A better way to synchronize the data transfer with notification of data arrival is to request an interrupt in the interrupt control field of the data transfer command packet. Then, when the data transfer command packet is removed from TERMQ, the application program can insert a write control message command packet onto INPTQ to notify the far-end DR device that the data transfer has completed.

Another consequence of command packet prefetching occurs, for example, when two write device command packets are inserted onto INPTQ. While the first data transfer takes place, the second command packet is prefetched and decoded. If an unusual event occurs and the application program must send an immediate control message to the far-end DR device, the application program might insert a write device control message packet onto INPTQ. However, this packet is not sent immediately because the second write device command packet has already been prefetched; the control message is sent after the second data transfer starts.

If the application program must send a control message with minimum delay, use one of the following techniques:

- Insert only one data transfer function onto INPTQ at a time. If this is done, a second transfer function will not be prefetched and a control message can be sent at any time.
- Use smaller buffers or a faster data rate to reduce the time necessary to complete a given command packet.
- Issue a Cancel I/O on Channel ($CANCEL) system service call followed by another IO$_STARTDATA function.

4.6.2 Action Routines

Action routines provide a useful DR32 programming technique. They can be used in application programs written in either assembly language or a high-level language. When a command packet is built, the address of a routine to be executed when the packet is removed from TERMQ is appended to the end of the packet. Then, rather than having to determine what action to perform for a particular packet when it is removed from TERMQ, the specified action routine is called.

4.6.3 Error Checking

Bits 0 through 23 in the second longword of the I/O status block correspond to the same bits in the DR32 status longword (DSL). Although the I/O status block is written only after the QIO function completes, the DSL is stored in every command packet. However, because there is no command packet in which to store a DSL for certain error conditions, such as FREEQ empty, some errors are reported only in the I/O status block. To check for an error under these conditions, examine the DSL in each packet for success or failure only. Then, if a failure occurs, the specific error can be determined from the I/O status block. The I/O status block should also be checked to verify that the QIO has not completed prior to a wait for the insertion of additional command packets onto TERMQ. In this way, the application program can detect asynchronous errors for which there is no command packet available.

4.6.4 Queue Retry Macro

When an interlocked queue instruction is included in the application program, the code should perform a retry if the queue is locked. However, the code should not execute an indefinite number of retries. Consequently, all retry loops should contain a maximum retry count. The macro programming example provided in Section 4.7 contains a useful queue retry macro.

4.6.5 Diagnostic Functions

The diagnostic functions listed in Table 4–2 can be used to test the DR32 without the presence of a far-end DR device. For the DR780, perform the following test sequence:

1. Insert a set self-test command packet onto INPTQ.
2. Insert a diagnostic write internal command packet that specifies a 128-byte buffer onto INPTQ. This packet copies 128 bytes from memory into the DR780 internal data silo.
3. Insert a diagnostic read DDI command packet onto INPTQ. This packet transmits the 128 bytes of data from the silo over the DDI and returns it to the silo.
4. Insert a diagnostic read internal command packet that specifies another 128-byte buffer in memory onto INPTQ. This packet copies 128 bytes of data from the silo into memory.
5. Compare the two memory buffers for equality. Note that on the DR780, the diagnostic read internal function destroys the first four bytes in the silo before storing the data in memory. Therefore, compare only the last 124 bytes of the two buffers.
6. Insert a clear self-test command packet onto INPTQ.

4.6.6 NOP Command Packet

It is often useful to insert a NOP command packet onto INPTQ to test the state of the DDI disable bit (XF$M_PKT_DDIDIS in the DSL). By checking this bit before initiating a data transfer, an application program can determine whether the far-end DR device is ready to accept data.

4.6.7 Interrupt Control Field

As described in Section 4.4.3.6, the interrupt control field determines the conditions under which an interrupt is generated: unconditionally, if TERMQ was empty, or never. The following are general applications of this field:

- If a program performs five data transfers and requires notification of completion only after all five have completed, the first four command packets should specify no interrupt and the fifth command packet should specify an unconditional interrupt.
- If a program performs a continuous series of data transfers, each command packet can specify an interrupt only if TERMQ was empty. Then, every time an event flag or AST notifies the program that a command packet was inserted onto TERMQ, the program removes and processes packets on TERMQ until it is empty.

- Command packets that specify no interrupt should never be mixed with command packets that specify an interrupt if TERMQ was empty.

4.7 Programming Examples

The programming examples in the following two sections use DR32 high-level language procedures and DR32 Queue I/O functions.

4.7.1 DR32 High-Level Language Program

The following sample program (Example 4–1) is an example of how the DR32 high-level language procedures perform a data transfer from a far-end DR device. The program reads a specified number of data buffers from an undefined far-end DR device, which is assumed to be a data source, into the VAX memory. The number of buffers is controlled by the NUMBUF parameter. The program contains examples of the read data chained function code and DR32 application program synchronization using AST routines and action routines.

Example 4–1 DR32 High-Level Language Program Example

```
************************************************************************
C
C                  DR32 HIGH-LEVEL LANGUAGE PROGRAM
C
************************************************************************

        INCLUDE 'XFDEF.FOR'                     ;DEFINE XF CONSTANTS
        PARAMETER       BUFSIZ = 1024           !SIZE OF EACH BUFFER
        PARAMETER       NUMBUF = 8              !NUMBER OF BUFFERS IN
                                                !RING
        PARAMETER       ILOGSIZ = 4             !SIZE OF INPUT LOG
                                                !ARRAY
        PARAMETER       EFN = 0                 !EVENT FLAG SYNCHRON-
                                                !IZING MAIN LEVEL WITH
                                                !AST ROUTINE
        INTEGER*2       BUFARRAY(BUFSIZ,NUMBUF) !THE RING OF BUFFERS
        INTEGER*2       INDEX                   !REFERS TO BUFFER
                                                !IN BUFARRAY
        INTEGER*2       COUNT                   !COUNTS NUMBER OF
                                                !BUFFERS FILLED
        INTEGER*2       DATART                  !DR32 CLOCK RATE

        INTEGER*4       CONTXT(30)      !CONTEXT ARRAY USED BY SUPPORT
        INTEGER*4       ILOGMSG(ILOGSIZ)!LOG MESSAGES FROM DEVICE
                                        !STORED HERE
        INTEGER*4       STATUS          !RETURNS FROM SUBROUTINES
        INTEGER*4       DEVMSG          !far-end DR device CODE

        EXTERNAL        ASTRTN          !AST ROUTINE
        EXTERNAL        AST$PROCBUF     !ACTION ROUTINE TO HANDLE
                                        !COMPLETION OF READ DATA
                                        !COMMAND PACKET

        EXTERNAL        AST$HALT        !ACTION ROUTINE TO HANDLE
                                        !COMPLETION OF A HALT
                                        !COMMAND PACKET

        COMMON  /MAIN_AST/      CONTXT, INDEX
        COMMON  /MAIN_ACTION/   BUFARRAY, ILOGMSG, COUNT
        EXTERNAL        SS$_NORMAL      !SUCCESS STATUS RETURN
```

(continued on next page)

Example 4–1 (Cont.) DR32 High-Level Language Program Example

```
*************************************************************************
C
C  THE CALL TO THE SETUP ROUTINE
C
*************************************************************************

        CALL XF$SETUP (CONTXT,BUFARRAY,BUFSIZ*2,NUMBUF,,,ILOGMSG,
        1               ILOGSIZ*4,,STATUS)
        IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))

C
C  PRELOAD THE INPUT QUEUE BEFORE STARTING THE DR32 IN ORDER TO AVOID
C  A DELAY IN THE DATA TRANSFER
C
C

*************************************************************************
C
C  BUILD COMMAND PACKETS
C
*************************************************************************

C  BUILD THE COMMAND PACKET THAT WILL INSTRUCT THE far-end DR device
C  TO START SAMPLING.  ARBITRARILY ASSUME THAT THE far-end DR device
C  WILL RECOGNIZE THIS DEVICE MESSAGE.  INSERT THIS PACKET ON THE
C  INPUT QUEUE (INPTQ).
C
        DEVMSG = 25                     !SIGNAL far-end DR device
                                        !"GO"
        CALL XF$PKTBLD (
        1       CONTXT,                 !THE CONTEXT ARRAY
        1       XF$K_PKT_WRTCM,         !WRITE CONTROL MESSAGE
                                        !FUNCTION
        1       ,,                      !NO INDEX OR SIZE
        1       DEVMSG,                 !SIGNAL "GO"

        1       4,                      !SIZE OF DEVMSG IN BYTES
        1       ILOGSIZ*4               !SPACE FOR INPUT LOG
                                        !MESSAGE
        1       XF$K_PKT_UNCOND         !MODES: UNCONDITIONAL
                                        !       INTERRUPT
        1       + XF$K_PKT_CBDM         !     : SEND FUNC AND DEVMSG
        1       + XF$K_PKT_INSTL        !     : INSERT PACKET AT INPTQ
                                        !       TAIL
        1       ''                      !NO ACTION ROUTINE OR ACTPARM
        1       STATUS)
        IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))

C
C  IN A LOOP, BUILD THE COMMAND PACKETS THAT WILL PERFORM THE CHAINED
C  READ TO INITIALLY FILL THE BUFFERS
C
```

(continued on next page)

Example 4–1 (Cont.) DR32 High-Level Language Program Example

```
        DO 10   INDEX = 1, NUMBUF       !FOR ALL BUFFERS DO
                CALL XF$PKTBLD(
        1       CONTXT,                 !THE CONTEXT ARRAY
        1       XF$K_PKT_RDCHN,         !READ DATA CHAINED
        1       INDEX,                  !IDENTIFIES BUFFER
        1       ,,,                     !NO SIZE, DEVMSG, OR DEVSIZ
        1       ILOGSIZ*4,              !SPACE FOR INPUT LOG MESSAGE
        1       XF$K_PKT_UNCOND         !MODES: UNCONDITIONAL
                                        !       INTERRUPT
        1       + XF$K_PKT_CB           !     : SEND FUNCTION CODE
        1       + XF$K_PKT_INSTL,       !     : INSERT PACKET AT INPTQ
                                        !       TAIL
        1       AST$PROCBUF,            !ACTION ROUTINE
        1       ,                       !NO ACTPARM
        1       STATUS)
        IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))
10      CONTINUE

C
C  THE INPUT QUEUE IS LOADED
C
*************************************************************************
C
C  START THE DR32
C
*************************************************************************

        DATART = 0                      !DATA TRANSFER RATE
        COUNT = 0                       !NUMBER OF BUFFERS THAT HAVE
                                        !BEEN FILLED
        CALL SYS$CLREF (%VAL(EFN))      !CLEAR EVENT FLAG BEFORE START

        CALL XF$STARTDEV (CONTXT,'XFA0:',ASTRTN,,,,DATART,STATUS)
        IF (STATUS .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP(%VAL(STATUS))

C
C  FROM THIS POINT, ROUTINES AT THE AST LEVEL ASSUME CONTROL.  WAIT
C  FOR THEM TO SIGNAL COMPLETION OF THE SAMPLING SWEEP.
C
        CALL SYS$WAITFR (%VAL(EFN))

        STOP
        END

*************************************************************************
C
C  AST ROUTINES
C
*************************************************************************

        SUBROUTINE      ASTRTN (ASTPARM)

        INCLUDE 'XFDEF.FOR/NOLIST'
        INTEGER*2       ASTPARM                 !UNUSED PARAMETER

        INTEGER*4       CONTXT(30)              !CONTEXT ARRAY
        INTEGER*4       STATUS                  !FOR CALL TO XF$GETPKT

        LOGICAL*1       WAITFLG                 !INPUT TO XF$GETPKT
        LOGICAL*1       LOGFLAG                 !INPUT TO XF$GETPKT

        COMMON  /MAIN_AST/      CONTXT, INDEX
```

(continued on next page)

Example 4–1 (Cont.) DR32 High-Level Language Program Example

```
        EXTERNAL        SS$_NORMAL
C
C  CALL XF$GETPKT IN A LOOP UNTIL TERMQ IS EMPTY.  XF$GETPKT WILL CALL
C  THE APPROPRIATE ACTION ROUTINE FOR EACH COMMAND PACKET.
C
        WAITFLG = .TRUE.                !DO NOT WAIT FOR EVENT FLAG
        LOGFLAG = .TRUE.                !REQUEST NOTIFICATION IF LOG
                                        !MESSAGE IS IN PACKET

10      CALL XF$GETPKT (CONTXT,WAITFLG,,INDEX,,LOGFLAG,STATUS)
        IF (STATUS .EQ. %LOC(SS$_NORMAL))      !PACKET FROM TERMQ
        1       GOTO 10
        IF (STATUS .EQ. SHR$_QEMPTY)    !TERMQ EMPTY - TRANSFER
        1       GOTO 20                 !STILL IN PROGRESS
        IF (STATUS .EQ. SHR$_HALTED .OR. STATUS .EQ. SHR$_NOCMDMEM)
        1       GOTO 20                 !TRANSFER COMPLETE.  NO MORE
                                        !COMMAND PACKETS. ASTS MAY
                                        !STILL BE DELIVERED

        CALL LIB$STOP (%VAL(STATUS))    !ERROR IN XF$GETPKT

20      RETURN
        END

*************************************************************************
C
C  ACTION ROUTINE
C
*************************************************************************

        SUBROUTINE      AST$PROCBUF (CONTXT,ACTPARM,DEVFLAG,LOGFLAG,
        1                               FUNC,INDEX,STATUS)

C
C  THIS IS THE ACTION ROUTINE CALLED BY XF$GETPKT WHEN IT REMOVES A
C  COMMAND PACKET FROM TERMQ.  THIS PACKET HAS JUST COMPLETED A READ
C  DATA OPERATION FROM THE BUFFER SPECIFIED BY INDEX.  THE BUFFER IS
C  PROCESSED, AND IF MORE DATA IS REQUIRED, THAT IS, BUFCOUNT .LE.
C  MAXCOUNT), ANOTHER PACKET IS BUILT.  THE BUFFER IN THIS PACKET IS
C  THEN REFILLED AND THE PACKET IS INSERTED ONTO INPTQ.
C  IF BUFCOUNT .GT. MAXCOUNT, THE SAMPLING SWEEP IS FINISHED AND A
C  HALT PACKET IS INSERTED ONTO INPTQ.
C
        INCLUDE         'XFDEF.FOR/NOLIST'
        PARAMETER       MAXCOUNT = 10   !NUMBER OF BUFFERS IN SWEEP
        PARAMETER       ILOGSIZ = 4     !SIZE OF INPUT LOG MESSAGE ARRAY
        PARAMETER       BUFSIZ = 1024   !SIZE OF EACH BUFFER (IN WORDS)
        PARAMETER       NUMBUF = 8      !NUMBER OF BUFFERS

        INTEGER*2       INDEX           !REFERS TO A BUFFER IN BUFARRAY
        INTEGER*2       FUNC            !FUNCTION CODE FROM PACKET
        INTEGER*2       BUFCOUNT        !COUNTS NUMBER OF BUFFERS FILLED
        INTEGER*2       BUFARRAY(BUFSIZ,NUMBUF) !THE ARRAY OF BUFFERS
        INTEGER*4       ACTPARM         !ACTION PARAMETER (NOT USED)
        INTEGER*4       STATUS          !STATUS OF XF$GETPKT (NOT USED)
        INTEGER*4       STAT            !STATUS OF CALL TO XF$PKTBLD
        INTEGER*4       CONTXT(30)      !CONTEXT ARRAY USED BY SUPPORT
        INTEGER*4       ILOGMSG(ILOGSIZ)!STORES LOG MESSAGES FROM DEVICE
        LOGICAL*1       DEVFLAG         !NOT USED IN THIS EXAMPLE
        LOGICAL*1       LOGFLAG         !SIGNALS LOG MESSAGE PRESENT

        COMMON  /MAIN_ACTION/   BUFARRAY,ILOGMSG,BUFCOUNT
```

(continued on next page)

Example 4–1 (Cont.) DR32 High-Level Language Program Example

```
        EXTERNAL        SS$_NORMAL
        EXTERNAL        AST$HALT
C
C  PROCESS THE BUFFER
C
        DO 10   I = 1, BUFSIZ
*************************************************************************
C
C  AT THIS POINT INSERT THE CODE TO PROCESS ELEMENT (I,INDEX) OF
C  BUFARRAY
C
*************************************************************************
10      CONTINUE

*************************************************************************
C
C  AT THIS POINT INSERT THE CODE TO LOOK AT THE LOG MESSAGE
C
*************************************************************************

C
C  IS THIS THE LAST BUFFER IN THE SWEEP?
C
BUFCOUNT = BUFCOUNT + 1
        IF (BUFCOUNT .LT. MAXCOUNT) THEN        !BUILD A PACKET TO
                                        !REFILL THE BUFFER
        CALL FAKE$PKTBLD (              !NEED INTERVENING ROUTINE
        1        CONTXT,                !THE CONTEXT ARRAY
        1       XF$K_PKT_RDCHN,         !READ DATA CHAINED
        1       INDEX,                  !BUFFER INDEX
        1       ,,,                     !NO SIZE, DEVMSG, OR DEVSIZ
        1       ILOGSIZ*4,              !SPACE FOR LOG MESSAGE
        1       XF$K_PKT_UNCOND         !MODES: UNCONDITIONAL
                                        !       INTERRUPT
        1       + XF$K_PKT_CB           !     : SEND CONTROL BYTE
        1       + XF$K_PKT_INSTL,       !     : INSERT AT TAIL
        1       ,,                      !ACTION GIVEN IN FAKE$PKTBLD
        1       STAT)
        IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))
        ELSE IF (BUFCOUNT .EQ. MAXCOUNT) THEN   !END OF CHAIN
                CALL FAKE$PKTBLD (      !NEED INTERVENING ROUTINE
        1       CONTXT,                 !THE CONTEXT ARRAY
        1       XF$K_PKT_RD,            !READ DATA FUNCTION
        1       INDEX,                  !BUFFER INDEX
        1       ,,,                     !NO SIZE, DEVMSG, OR DEVSIZ
        1       ILOGSIZ*4,              !SPACE FOR LOG MESSAGE
        1       XF$K_PKT_UNCOND         !MODES: UNCONDITIONAL
                                        !       INTERRUPT
        1       + XF$K_PKT_CB           !     : SEND CONTROL BYTE
        1       + XF$K_PKT_INSTL,       !     : INSET AT TAIL
        1       ,,                      !ACTION GIVEN IN FAKE$PKTBLD
        1       STAT)
        IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))
        ELSE                            !BUILD A HALT PACKET
                CALL XF$PKTBLD (
        1       CONTXT,                 !THE CONTEXT ARRAY
        1       XF$K_PKT_HALT,          !ALL DONE
        1       ,,,,                    !DEFAULT VALUES
        1       ILOGSIZ*1,              !SPACE FOR INPUT LOG MESSAGE
```

(continued on next page)

Example 4–1 (Cont.) DR32 High-Level Language Program Example

```
       1       AST$HALT,                !ACTION ROUTINE
       1       ,                        !NO ACTPARM
       1       STAT)
       IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))

       END IF

       RETURN
       END

*************************************************************************
C
C  PASS ADDRESS OF ACTION ROUTINE TO COMMAND PACKET
C
*************************************************************************

       SUBROUTINE      FAKE$PKTBLD(A,B,C,D,E,F,G,H,I,J,K)

C
C  AST$PROCBUF CALLS THIS SUBROUTINE IN ORDER TO PASS THE ADDRESS OF
C  AST$PROCBUF TO XF$PKTBLD.  (AST$PROCBUF CANNOT REFER TO ITSELF
C  WITHIN THE SCOPE OF AST$PROCBUF)
C

       EXTERNAL        AST$PROCBUF

       CALL XF$PKTBLD (A,B,C,D,E,F,G,H,AST$PROCBUF,J,K)

       RETURN
       END
*************************************************************************
C
C  HALT ACTION ROUTINE
C
*************************************************************************

       SUBROUTINE      AST$HALT (CONTXT,ACTPARM,DEVFLAG,LOGFLAG,
                                FUNC,INDEX,STATUS)
C
C  THIS IS THE ACTION ROUTINE CALLED BY XF$GETPKT WHEN IT REMOVES A
C  HALT PACKET FROM TERMQ.  THIS ROUTINE PRINTS STATUS INFORMATION,
C  CALLS XF$CLEANUP TO PERFORM FINAL HOUSEKEEPING FUNCTIONS, AND SETS
C  THE EVENT FLAG THAT SIGNALS THE TRANSFER IS COMPLETE.
C
       PARAMETER       EFN = 0

       INTEGER*2       FUNC             !NOT USED
       INTEGER*2       INDEX            !NOT USED

       INTEGER*4       ACTPARM          !NOT USED
       INTEGER*4       STATUS           !NOT USED
       INTEGER*4       STAT             !RETURN FROM XF$CLEANUP
       INTEGER*4       CONTXT(30)       !CONTEXT ARRAY USED BY SUPPORT

       LOGICAL*1       DEVFLAG          !NOT USED
       LOGICAL*1       LOGFLAG          !SIGNALS LOG MESSAGE

       EXTERNAL        SS$_NORMAL       !SUCCESS STATUS RETURN
C
C  PRINT FINAL STATUS
C
       PRINT *,  'FINAL STATUS IN I/O STATUS BLOCK'
       PRINT *,  CONTXT(1), CONTXT(2)
```

(continued on next page)

Example 4–1 (Cont.) DR32 High-Level Language Program Example

```
C
C  CLEAN UP
C
        CALL XF$CLEANUP (CONTXT,STAT)
        IF (STAT .NE. %LOC(SS$_NORMAL)) CALL LIB$STOP (%VAL(STAT))

        CALL SYS$SETEF (%VAL(EFN))

        RETURN
        END
```

4.7.2 DR32 Queue I/O Functions Program

The following sample program (Example 4–2) uses Queue I/O functions to send a device message to the far-end DR device and then waits for a message returned in a command packet on FREEQ. The returned message is copied into another command packet, and that packet writes a data buffer to the far-end DR device.

Example 4–2 DR32 Queue I/O Functions Program Example

```
; **********************************************************************
;
;
;                 DR32 QUEUE I/O FUNCTIONS PROGRAM
;
;
; **********************************************************************

        .TITLE  DR32 PROGRAMMING EXAMPLE
        .IDENT  /01/

;
; DEFINE SYMBOLS
;
        $XFDEF
;
;
; QRETRY - THIS MACRO EXECUTES AN INTERLOCKED QUEUE INSTRUCTION AND
;          RETRIES THE INSTRUCTION UP TO 25 TIMES IF THE QUEUE IS
;          LOCKED.
;
```

(continued on next page)

Example 4–2 (Cont.) DR32 Queue I/O Functions Program Example

```
; INPUTS:
;
;       OPCODE = OPCODE NAME: INSQHI,INSQTI,REMQHI,REMQTI
;       OPERAND1 = FIRST OPERAND FOR OPCODE
;       OPERAND2 = SECOND OPERAND FOR OPCODE
;       SUCCESS = LABEL TO BRANCH TO IF OPERATION SUCCEEDS
;       ERROR = LABEL TO BRANCH TO IF OPERATION FAILS
;
; OUTPUTS:
;
;       R0 = DESTROYED
;
;       C-BIT = CLEAR IF OPERATION SUCCEEDED
;               SET IF OPERATION FAILED - QUEUE LOCKED
;               (MUST BE CHECKED BEFORE V-BIT OR Z-BIT)
;
;       REMQTI OR REMQHI:
;
;               V-BIT = CLEAR IF AN ENTRY REMOVED FROM QUEUE; SET
;                       IF NO ENTRY REMOVED FROM QUEUE.
;
;       INSQTI OR INSQHI:
;
;               Z-BIT = CLEAR IF ENTRY IS NOT FIRST IN QUEUE; SET
;                       IF ENTRY IS FIRST IN QUEUE.
;
        .MACRO  QRETRY  OPCODE,OPERAND1,OPERAND2,SUCCESS,ERROR,?LOOP,
                        ?OK
        CLRL    R0
LOOP:
        OPCODE  OPERAND1,OPERAND2
        .IF     NB      SUCCESS
        BCC     SUCCESS
        .IFF
        BCC     OK
        .ENDC
        AOBLSS  #25,R0,LOOP
        .IF     NB      ERROR
        BRW     ERROR
        .ENDC
OK:
        .ENDM   QRETRY


;
; ALLOCATE STORAGE FOR DATA STRUCTURES
;
        .PSECT  DATA,QUAD
CMDBLK:                                 ; COMMAND BLOCK
```

(continued on next page)

Example 4–2 (Cont.) DR32 Queue I/O Functions Program Example

```
INPTQ:  .BLKQ   1                       ; INPUT QUEUE
TERMQ:  .BLKQ   1                       ; TERMINATION QUEUE
FREEQ:  .BLKQ   1                       ; FREE QUEUE
MSGPKT:                                 ; THIS PACKET SENDS A 12-BYTE
                                        ; DEVICE MESSAGE
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   12                      ; LENGTH OF DEVICE MESSAGE
        .BYTE   0                       ; LENGTH OF LOG AREA
        .BYTE   XF$K_PKT_WRTCM          ; COMMAND  = WRITE CONTROL
                                        ; MESSAGE
        .BYTE   XF$K_PKT_NOINT@-        ; PACKET CONTROL = NO
                                        ; INTERRUPT
                XF$V_PKT_INTCTL
        .BLKL   1                       ; BYTE COUNT
        .BLKL   1                       ; BUFFER ADDRESS
        .BLKL   2                       ; RESIDUAL MEMORY AND DDI BYTE
                                        ; COUNTS
        .BLKL   1                       ; DR32 STATUS LONGWORD
        .LONG   11111,22222,33333       ; DEVICE MESSAGE
        .LONG   0                       ; EXTEND DEVICE MESSAGE TO
                                        ; QUADWORD LENGTH
        .ALIGN  QUAD

WRTPKT:                                 ; THIS PACKET DOES A WRITE
                                        ; DEVICE
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   4                       ; LENGTH OF DEVICE MESSAGE
        .BYTE   0                       ; LENGTH OF LOG AREA
        .BYTE   XF$K_PKT_WRT            ; COMMAND = WRITE
        .BYTE   <XF$K_PKT_CBDMBC@-      ; PACKET CONTROL = SEND
                                        ; COMMAND BYTE,
                XF$V_PKT_CISEL>!-       ; DEVICE MESSAGE, AND BYTE
                                        ; COUNT
                <XF$K_PKT_NOINT@-       ; AND NO INTERRUPT
                XF$V_PKT_INTCTL>
        .LONG   1000                    ; BYTE COUNT
        .LONG   WRTBFR                  ; BUFFER ADDRESS
        .BLKL   2                       ; RESIDUAL MEMORY AND DDI BYTE
                                        ; COUNTS
        .BLKL   1                       ; DR32 STATUS LONGWORD

WDVMSG: .BLKQ   1                       ; SPACE FOR DEVICE MESSAGE

        .ALIGN  QUAD
HLTPKT:                                 ; THIS PACKET HALTS THE DR32
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   0,0,XF$K_PKT_HALT,0     ; COMMAND = HALT
        ,BLKL   5                       ; UNUSED FIELDS IN THIS PACKET

        .ALIGN QUAD
FREPKT:                                 ; PACKET FOR FREE QUEUE
        .BLKQ   1                       ; QUEUE LINKS
        .BYTE   4,0,0,0                 ; LENGTH OF DEVICE MESSAGE
                                        ; FIELD
        .BLKL   4                       ; UNUSED FIELDS IN THIS PACKET
        .BLKL   1                       ; DR32 STATUS LONGWORD
        .BLKQ   1                       ; SPACE FOR DEVICE MESSAGE
CMDBLKSIZ=.-CMDBLK

BFRBLK:                                 ; BUFFER BLOCK

WRTBFR: .BLKB   1000
```

(continued on next page)

Example 4–2 (Cont.) DR32 Queue I/O Functions Program Example

```
BFRBLKSIZ=.-BFRBLK

CMDTBL: .LONG   CMDBLKSIZ               ; COMMAND BLOCK SIZE
        .LONG   CMDBLK                  ; COMMAND BLOCK ADDRESS
        .LONG   BFRBLKSIZ               ; BUFFER BLOCK SIZE
        .LONG   BFRBLK                  ; BUFFER BLOCK ADDRESS
        .LONG   PKTAST                  ; PACKET AST ADDRESS
        .LONG   0                       ; PACKET AST PARAMETER
        .BYTE   236,XF$M_CMT_SETRTE,0,0 ; DATA RATE (2.0 MBYTES/SEC)
        .LONG   GOBITADR                ; ADDRESS TO STORE THE GO
                                        ; BIT ADDRESS
GOBITADR:
        .BLKL   1

XFIOSB: .BLKL   2                       ; I/O STATUS BLOCK

XFNAMEDSC:
        .LONG   XFNAMESIZ               ; NAME DESCRIPTOR
        .LONG   XFNAME

XFCHAN: .BLKW   1                       ; CHANNEL NUMBER

XFNAME: .ASCII  /XFA0/
XFNAMESIZE=.-XFNAME

; *********************************************************************
;
; PROGRAM STARTING POINT
;
; *********************************************************************

        .PSECT  CODE,NOWRT

        .ENTRY  DREXAMPLE,M<R2,R3>

        $ASSIGN_S DEVNAM = XFNAMEDSC,-  ; ASSIGN A CHANNEL TO DR32
                  CHAN = XFCHAN
        BLBS    R0,10$                  ; SUCCESSFUL ASSIGN
        BRW     ERROR
10$:    MOVAB   CMDBLK,R2
        CLRQ    (R2)+                   ; INITIALIZE INPTQ
        CLRQ    (R2)+                   ; INITIALIZE TERMQ
        CLRQ    (R2)                    ; INITIALIZE FREEQ
;
; INSERT COMMAND PACKET ONTO FREEQ FOR RETURN MESSAGE
;

        QRETRY  ERROR=BADQUEUE,-
        INSQTI  FREPKT,FREEQ

;
; START DEVICE
;

        $QIO_S  FUNC = #IO$_STARTDATA,-
                CHAN = XFCHAN,-
                IOSB = XFIOSB,-
                EFN = #1,-
                P1 = CMDTBL,-
                P2 = #XF$K_CMT_LENGTH
        BLBC    R0,ERROR
;
; SEND MESSAGE TO far-end DR device
;
```

(continued on next page)

Example 4–2 (Cont.) DR32 Queue I/O Functions Program Example

```
        QRETRY  ERROR=BADQUEUE,-
        INSQTI  MSGPKT,INPTQ
        MOVL    #1,@GOBITADR            ; SET GO BIT
        $WAITFR_S #1                    ; WAIT UNTIL QIO COMPLETES
;
; CHECK FOR SUCCESSFUL COMPLETION
;
        MOVZWL  XFIOSB,R0
        BEQL    BADQUEUE                 ; I/O NOT DONE YET - BAD QUEUE
                                         ; ERROR IN AST ROUTINE
        BLBC    R0,ERROR                 ; ERROR
        RET                              ; SUCCESSFUL COMPLETION

BADQUEUE:
        MOVZWL  #SS$_BADQUEUEHDR,R0

;
; AN ERROR HAS OCCURRED.  NORMALLY, YOU MIGHT PERFORM MORE
; EXTENSIVE ERROR CHECKING AT THIS POINT.  IN PARTICULAR, IF THE ERROR
; IS SS$_CTRLERR, SS$_DEVREQERR, OR SS$_PARITY, THE SECOND LONGWORD
; OF THE I/O STATUS BLOCK CAN PROVIDE ADDITIONAL INFORMATION.  IN THIS
; EXAMPLE, THE PROGRAM EXITS WITH THE ERROR STATUS IN R0.
;
;
; COMMAND PACKET AST ROUTINE
;

PKTAST: .WORD   0
NXTPKT: QRETRY  ERROR=70$,-              ; GET NEXT PACKET FROM QUEUE
        REMQHI  TERMQ,R1
        BVC     10$                      ; PACKET OBTAINED FROM QUEUE
        RET                              ; QUEUE IS EMPTY
10$:    BLBC    XF$L_PKT_DSL(R1),50$     ; RETURN IF PACKET ERROR
        BBC     #XF$V_PKT_FREQPK,-       ; RETURN IF PACKET NOT FROM
                XF$L_PKT_DSL(R1),50$     ; FREEQ

;
; COMMAND PACKET OBTAINED FROM FREEQ.  COPY DEVICE MESSAGE AND QUEUE
; WRITE PACKET.
;

        MOVL    XF$B_PKT_DEVMSG(R1),WDVMSG
        QRETRY  ERROR=70$,-
        INSQTI  WRTPKT,INPTQ
        QRETRY  ERROR=70$,-
        INSQTI  HLTPKT,INPTQ
        MOVL    #1,@GOBITADR             ; SET GO BIT
50$:    RET

;
; BAD QUEUE ERROR IN AST ROUTINE - WAKE UP MAIN LEVEL.  QIO MAY
; OR MAY NOT HAVE COMPLETED.
;

70$:    $SETEF_S  #1                     ; WAKE UP MAIN LEVEL
        RET

        .END    DREXAMPLE
```

5
Asynchronous DDCMP Interface Driver

This chapter describes the use of the asynchronous DDCMP interface driver in an OpenVMS VAX environment.

5.1 Supported Devices

Asynchronous DDCMP is supported for DECnet for OpenVMS using software DDCMP over terminal ports. This enables all Digital supported terminal devices to provide a DDCMP interface between two VAX processors using terminal ports. Asynchronous DDCMP supports full-duplex, point-to-point lines.

5.2 Driver Features and Capabilities

The asynchronous DDCMP driver provides the following capabilities:

- Point-to-point operating mode in which the asynchronous DDCMP port is connected to one other controller also operating in point-to-point mode
- A nonprivileged QIO interface to the asynchronous DDCMP for using this device as a raw-data channel
- Full-duplex operation
- Interface design common to all communications devices supported by the OpenVMS VAX operating system
- Separate transmit and receive queues
- Assignment of multiple read and write buffers to the device

5.2.1 Quotas

Transmit operations are buffered and I/O operations and are limited by the process's buffered I/O quota.

The quotas for the receive buffer free list are the process's buffered I/O quota and buffered I/O byte count quota.

5.2.2 Power Failure

If a system power failure occurs, no asynchronous DDCMP recovery is possible. The driver is in a fatal error state and shuts down.

5.3 Device Information

You can obtain information about asynchronous DDCMP characteristics by using the Get Device/Volume Information ($GETDVI) system service. (See the *OpenVMS System Services Reference Manual*.)

$GETDVI returns device characteristics when you specify the item code DVI$_DEVCHAR. Table 5–1 lists these characteristics, which are defined by the $DEVDEF macro.

Table 5–1 Device Characteristics

Characteristic[1]	Meaning
	Static Bits (Always Set)
DEV$M_NET	Network device. Set for terminal port if it is a network device.
DEV$M_AVL	Available device. Set when unit control block (UCB) is initialized.
DEV$M_ODV	Output device.
DEV$M_IDV	Input device.

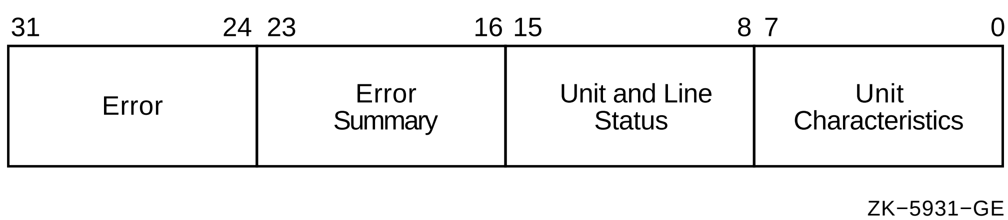
[1]Defined by the $DEVDEF macro

DVI$_DEVCLASS returns the device class, which is DC$_SCOM. DVI$_DEFTYPE returns the device type, which is the terminal ports device type. The $DCDEF macro defines the device class and device type names.

DVI$_DEVBUFSIZ returns the maximum message size. The maximum message size is the maximum send or receive message size for the unit. Messages greater than 512 bytes on modem-controlled lines are more prone to transmission errors.

DVI$_DEVDEPEND returns the unit characteristics bits, the unit and line status bits, the error summary bits, and the specific errors in a longword field as shown in Figure 5–1.

Figure 5–1 DVI$_DEVDEPEND Returns



Unit characteristics bits govern the DDCMP operating mode. They are defined by the $XMDEF macro and can be set by a set mode function (see Section 5.4.3.1) or can be read by a sense mode function (see Section 5.4.4).

The status bits show the status of the unit and the line. These bits can be set or cleared only when the controller and tributary are not active.

Table 5–2 lists the status values and their meanings. The values are defined by the $XMDEF macro.

Table 5–2 Asynchronous DDCMP Unit and Line Status

Status	Meaning
XM$M_STS_ACTIVE	DDCMP protocol is active.
XM$M_STS_DISC	Modem line went from on to off. This bit will be returned in the field IRP$L_IOST2 if the driver has had a timeout while waiting for the CTS signal to be present on the device.
XM$M_STS_BUFFAIL	Receive buffer allocation failed.

The error summary bits are set when an error occurs. They are read-only bits. If the error is fatal, the asynchronous DDCMP for that port is shut down. Table 5–3 lists the error summary bit values and their meanings.

Table 5–3 Error Summary Bits

Error Summary Bit	Meaning
XM$M_ERR_MAINT	DDCMP maintenance message received
XM$M_ERR_START	DDCMP start message received
XM$M_ERR_FATAL	Hardware or software error occurred on controller
XM$M_ERR_TRIB	Hardware or software error occurred on tributary
XM$M_ERR_LOST	Data lost when a received message was longer than the specified maximum message size
XM$M_ERR_THRESH	Receive, transmit, or select threshold errors

Table 5–4 lists the errors that can be specified. These errors are mapped to the indicated codes.

Table 5–4 Asynchronous DDCMP Errors

Value (octal)	Meaning	Code Set
2	Receive threshold error	XM$M_ERR_THRESH
4	Transmit threshold error	XM$M_ERR_THRESH
6	Select threshold error	XM$M_ERR_THRESH
10	Start received in run state	XM$M_ERR_START
12	Maintenance received in run state	XM$M_ERR_MAINT
14	Maintenance received in halt state	(none)
16	Start received in maintenance state	XM$M_ERR_START
100–276	Internal procedure (software) errors	XM$M_ERR_TRIB
300	Buffer too small	XM$M_ERR_LOST
302	Nonexistent memory	XM$M_ERR_FATAL
304	Modem disconnected	XM$M_STS_DISC

5.4 Asynchronous DDCMP Function Codes

The asynchronous DDCMP driver can perform logical, virtual, and physical I/O operations. The basic functions are read, write, set mode, set characteristics, and sense mode. Table 5–5 lists these functions and their function codes. The sections that follow describe these functions in greater detail.

Table 5–5 Asynchronous DDCMP I/O Functions

Function Code	Arguments	Type[1]	Modifiers	Function
IO$_READLBLK	P1,P2	L	IO$M_NOW	Read logical block.
IO$_READVBLK	P1,P2	V	IO$M_NOW	Read virtual block.
IO$_READPBLK	P1,P2	P	IO$M_NOW	Read physical block.
IO$_WRITELBLK	P1,P2	L		Write logical block.
IO$_WRITEVBLK	P1,P2	V		Write virtual block.
IO$_WRITEPBLK	P1,P2	P		Write physical block.
IO$_SETMODE	P1,[P2],P3	L	IO$M_CTRL IO$M_SHUTDOWN IO$M_STARTUP IO$M_ATTNAST	Set asynchronous DDCMP characteristics and controller state for subsequent operations.
IO$_SETCHAR	P1,[P2],P3	P	IO$M_CTRL IO$M_SHUTDOWN IO$M_STARTUP IO$M_ATTNAST	Set asynchronous DDCMP characteristics and controller state for subsequent operations.
IO$_SENSEMODE	P1,P2	L	IO$M_CTRL IO$M_CLR_COUNTS IO$M_RD_COUNTS	Sense controller or tributary characteristics and return them in specified buffers.

[1]V = virtual, L = logical, P = physical (there is no functional difference in these operations)

Although the asynchronous DDCMP driver does not differentiate among logical, virtual, and physical I/O functions (all are treated identically), you must have the required privilege to issue a request. (Logical I/O functions require no I/O privilege.)

5.4.1 Read

Read functions provide for the direct transfer of data into the user process's virtual memory address space. The operating system provides the following function codes:

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

Received messages are multibuffered in system dynamic memory and then copied to the user's buffer.

The read functions take the following device- or function-dependent arguments:

- P1—The starting virtual address of the buffer that is to receive data
- P2—The size of the receive buffer in bytes

The message size specified by P2 cannot be larger than the maximum receive-message size for the unit (see Section 5.3). If a message larger than the maximum size is received, a status of SS$_DATAOVERUN is returned in the I/O status block.

The read functions can take the following function modifier:

- IO$M_NOW—Complete the read operation immediately with a received message. (If no message is currently available, return a status of SS$_ENDOFFILE in the I/O status block.)

5.4.2 Write

Write functions provide for the direct transfer of data from the user process's virtual memory address space. The operating system provides the following function codes:

- IO$_WRITELBLK—Write logical block
- IO$_WRITEVBLK—Write virtual block
- IO$_WRITEPBLK—Write physical block

Asynchronous DDCMP messages are copied into a system buffer before they are transmitted.

The write functions take the following device- or function-dependent arguments:

- P1—The starting virtual address of the buffer containing the data to be transmitted
- P2—The size of the buffer in bytes

The message size specified by P2 cannot be larger than the maximum send-message size for the unit (see Section 5.3).

The write functions take no function modifiers.

5.4.3 Set Mode and Set Characteristics

Set mode operations are used to perform protocol, operational, and program and driver interface operations with the asynchronous DDCMP driver. The operating system defines the following types of set mode functions:

- Set mode
- Set characteristics
- Set controller mode
- Set tributary mode
- Enable attention AST
- Shutdown controller
- Shutdown tributary

Used without function modifiers, set mode and set characteristics functions can modify an existing tributary. Used with certain function modifiers, they can perform asynchronous DDCMP operations such as starting a tributary and requesting an attention AST. The operating system provides the following function codes:

- IO$_SETMODE—Set mode (no I/O privilege required)

- IO$_SETCHAR—Set characteristics (requires physical I/O privilege)

The other five types of set mode functions, which use the two function codes with certain function modifiers, are described in the sections that follow.

To use the IO$_SETMODE and IO$_SETCHAR functions, assign the appropriate unit control block (UCB) with the Assign I/O Channel ($ASSIGN) system service.

5.4.3.1 Set Controller Mode

The set controller mode function sets the asynchronous DDCMP controller state and activates the controller. The first occurrence of an IO$_SETMODE function creates a buffer for the driver to use. (Part of the buffer created by IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP is allocated for the protocol operation to use.) The following combinations of function code and modifier are provided:

- IO$_SETMODE!IO$M_CTRL—Set controller characteristics
- IO$_SETCHAR!IO$M_CTRL—Set controller characteristics
- IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP—Set controller characteristics and start the controller
- IO$_SETCHAR!IO$M_CTRL!IO$M_STARTUP—Set controller characteristics and start the controller

If the function modifier IO$M_STARTUP is specified, the controller is started and the modem is enabled. If IO$M_STARTUP is not specified, the specified characteristics are simply modified.

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

- P2—The address of a descriptor for a characteristics buffer (optional)

The P2 buffer consists of a series of six-byte entries. The first word contains the parameter identifier (ID), and the longword that follows contains one of the values that can be associated with the parameter ID. Figure 5–2 shows the format for this buffer.

Figure 5–2 P2 Characteristics Buffer (Set Controller)

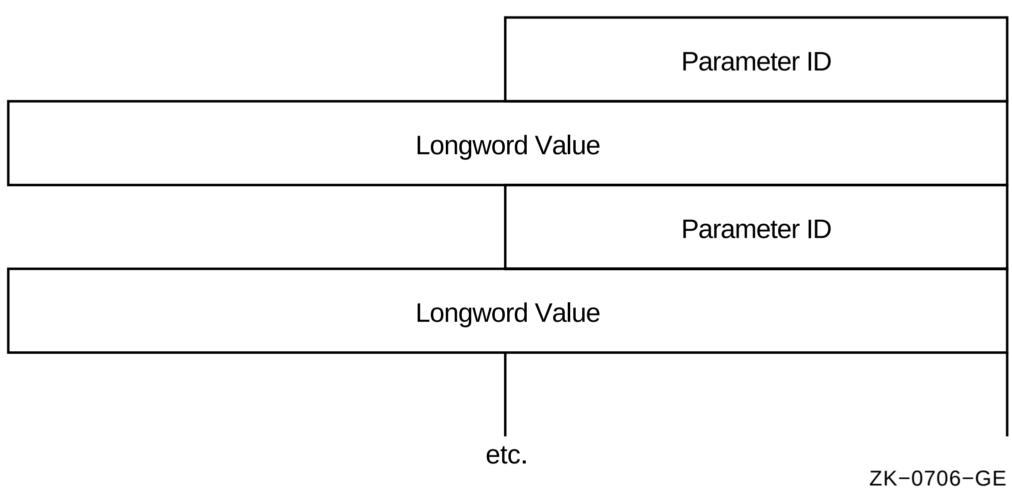


Table 5–6 lists the parameter IDs and values that can be specified in the P2 buffer. The $NMADEF macro defines these values.

Table 5–6 P2 Characteristics Values (Set Controller)

Parameter ID	Meaning
NMA$C_PCLI_PRO	Protocol mode. Only the following value can be specified: **Value** / **Meaning** NMA$C_LINPR_POI — DDCMP point-to-point (default)
NMA$C_PCLI_DUP	Duplex mode. Only the following value can be specified: **Value** / **Meaning** NMA$C_DPX_FUL — Full-duplex (default)
NMA$C_PCLI_CON	Controller mode. Only the following value can be specified: **Value** / **Meaning** NMA$C_LINCN_NOR — Normal (default)
NMA$C_PCLI_BFN	Number of receive buffers to preallocate.
NMA$C_PCLI_BUS	Maximum allowable transmit and receive message length (default = 512 bytes).

5.4.3.2 Set Tributary Mode

The set tributary mode function either starts a tributary or modifies an existing one. This function must be performed before any communication can occur with the attached unit.

Because the asynchronous DDCMP driver deals with only one tributary, the set tributary function starts both the tributary and the protocol. The data block that describes the tributary has already been created.

The operating system provides the following combinations of function code and modifier:

- IO$_SETMODE—Modify tributary characteristics
- IO$_SETCHAR—Modify tributary characteristics
- IO$_SETMODE!IO$M_STARTUP—Start tributary
- IO$_SETCHAR!IO$M_STARTUP—Start tributary

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

- P2—The address of a descriptor for a characteristics buffer (optional)

The P2 buffer consists of a series of six-byte entries. The first longword contains the parameter identifier (ID), and the longword that follows contains one of the values that can be associated with the parameter ID. Figure 5–2 shows the format for this buffer.

Table 5–7 lists the parameter IDs and values that can be specified in the P2 buffer.

Table 5–7 P2 Characteristics Values (Set Tributary)

Parameter ID	Meaning
NMA$C_PCCI_TRT[1]	Transmit delay timer (default = 0).
NMA$C_PCCI_RTT[1]	Retransmit timer for full-duplex point-to-point mode and selection timer for multipoint control and half-duplex point-to-point mode (default = 3000).

[1]A global polling parameter. All timer values must be specified in milliseconds.

On receipt of the QIO request for asynchronous DDCMP, the driver modifies the tributary parameters and starts the protocol. The tributary state and the protocol state are equal. The driver does not verify that a tributary address has been provided. If an address has not been provided, it defaults to 1.

5.4.3.3 Shutdown Controller

The shutdown controller function shuts down the controller and disables the modem line. On completion of a shutdown controller request, all tributaries have been halted (including those tributaries not explicitly halted), all tributary buffers returned, and the controller reinitialized. This function halts the tributary, the protocol, and the line. The controller cannot be used again until another IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP or IO$_SETCHAR!IO$M_CTRL!IO$M_STARTUP request has been issued (see Section 5.4.3.1).

The operating system provides the following combinations of function code and modifier:

- IO$_SETMODE!IO$M_CTRL!IO$M_SHUTDOWN—Shutdown controller
- IO$_SETCHAR!IO$M_CTRL!IO$M_SHUTDOWN—Shutdown controller

The shutdown controller function takes no device- or function-dependent arguments.

5.4.3.4 Shutdown Tributary

The shutdown tributary function halts, but does not delete, the specified tributary. On completion of a shutdown tributary request, the tributary and the protocol are halted, all buffers are returned, and all pending I/O requests and received messages are aborted. Neither the tributary nor the attached device can be used again until another IO$_SETMODE!IO$M_STARTUP or IO$_SETCHAR!IO$M_STARTUP request has been issued (see Section 5.4.3.2).

The operating system provides the following combinations of function code and modifier:

- IO$_SETMODE!IO$M_SHUTDOWN—Shutdown tributary
- IO$_SETCHAR!IO$M_SHUTDOWN—Shutdown tributary

The shutdown tributary function takes no device- or function-dependent arguments.

5.4.3.5 Enable Attention AST

The enable attention AST function requests that an attention AST be delivered to the requesting process when a status change occurs on the specified tributary. An AST is queued when the driver sets or clears either an error summary bit or any of the unit status bits (see Tables 5–2 and 5–3), or when a message is available and there is no waiting read request. The enable attention AST function is legal at any time, regardless of the condition of the unit status bits.

The operating system provides the following combinations of function code and modifier:

- IO$_SETMODE!IO$M_ATTNAST—Enable attention AST
- IO$_SETCHAR!IO$M_ATTNAST—Enable attention AST

These codes take the following device- or function-dependent arguments:

- P1—The address of an AST service routine or 0 for disable
- P2—Ignored
- P3—Access mode to deliver AST

The enable attention AST function enables an attention AST to be delivered to the requesting process once only. After the AST occurs, it must be explicitly reenabled by the function before the AST can occur again. The function is also subject to AST quotas.

The AST service routine is called with an argument list. The first argument is the current value of the second longword of the I/O status block (see Section 5.5). The access mode specified by P3 is maximized with the requester's access mode.

5.4.4 Sense Mode

The sense mode function returns the controller or tributary characteristics in the specified buffers.

The operating system provides the following function codes:

- IO$_SENSEMODE!IO$M_CTRL—Read controller characteristics
- IO$_SENSEMODE—Read tributary characteristics

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

- P2—The address of a descriptor for a buffer into which the characteristics buffer is stored (optional). (Figure 5–2 shows the format of the characteristics buffer.)

All characteristics that fit into the buffer specified by P2 are returned. However, if all the characteristics cannot be stored in the buffer, the I/O status block returns the status SS$_BUFFEROVF. The second word of the I/O status block returns the size (in bytes) of the characteristics buffer returned by P2 (see Section 5.5).

5.4.4.1 Read Internal Counters

The read internal counters (IO$M_RD_COUNTS) subfunction reads the DDCMP internal counters. The operating system provides the following combinations of function codes and modifiers:

- IO$_SENSEMODE!IO$M_RD_COUNTS—Read tributary counters
- IO$_SENSEMODE!IO$M_CLR_COUNTS—Clear tributary counters

- IO$_SENSEMODE!IO$M_RD_COUNTS!IO$M_CLR_COUNTS—Read and then clear tributary counters
- IO$_SENSEMODE!IO$M_CTRL!IO$M_RD_COUNTS—Read controller counters
- IO$_SENSEMODE!IO$M_CTRL!IO$M_CLR_COUNTS—Clear controller counters
- IO$_SENSEMODE!IO$M_CTRL!IO$M_RD_COUNTS!IO$M_CLR_COUNTS—Read and then clear controller counters

These codes take the following device- or function dependent arguments:

- P1—Ignored
- P2—The address of a buffer descriptor into which the counters will be returned

Figure 5–3 shows the format of the buffer. All counters that fit into the buffer specified by P2 are returned. However, if all the counters cannot be stored in the buffer, the I/O status block returns the status SS$_BUFFEROVF. The second word of the I/O status block returns the size, in bytes, of the extended characteristics buffer returned (see Section 5.5).

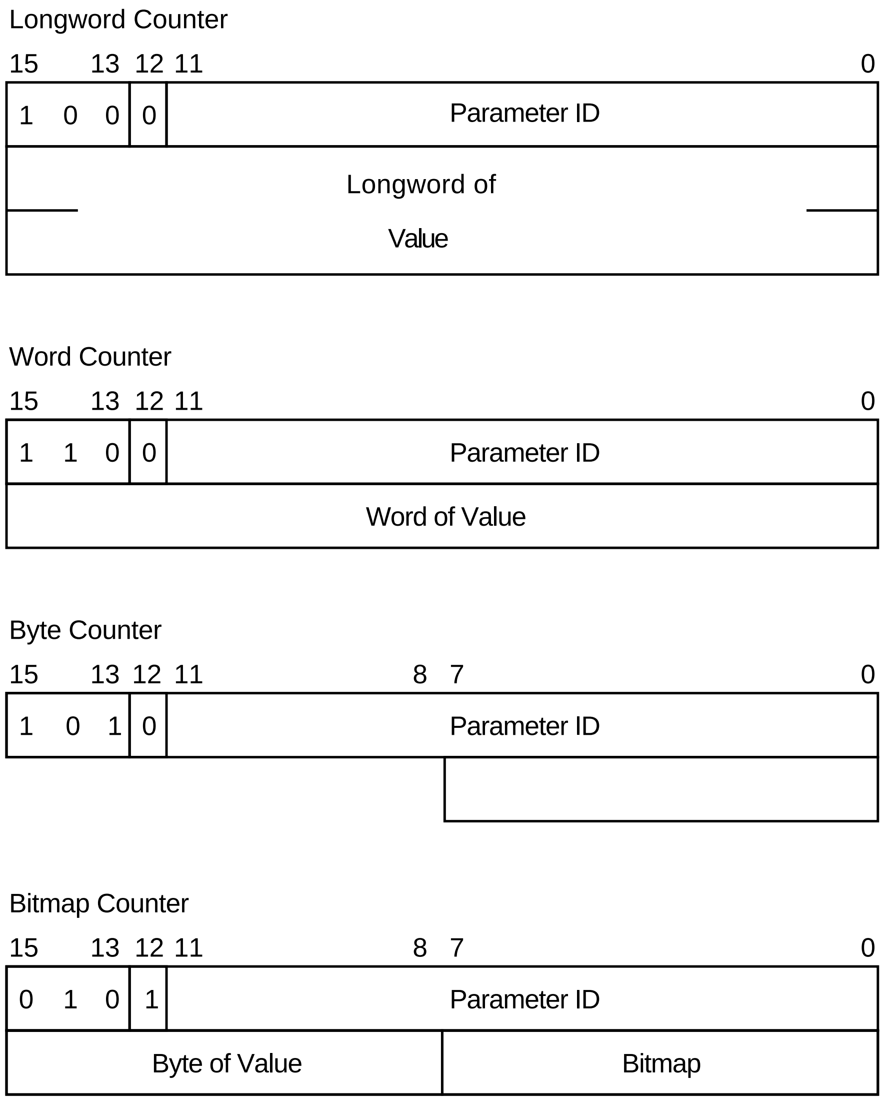
Table 5–8 lists the parameter IDs that can be returned for asynchronous DDCMP.

Table 5–8 Controller Counter Parameter IDs

Parameter ID	Meaning	
NMA$C_CTLIN_LPE	Number of local station errors bitmap counter.	
	Value	**Meaning**
	1	Receive overrun SNAK set.
	2	Receive overrun SNAK not set.
	4	Transmitter underrun.
	8	Message format error.
NMA$C_CTLIN_RPE	Number of remote station errors bitmap counter.	
	Value	**Meaning**
	1	NAKs received due to receiver overrun.
	2	NAKs received due to message format error.
	4	SNAK set message format error.
	8	Streaming tributary.

Table 5–9 lists the parameter IDs that can be returned for tributaries.

Figure 5–3 P2 Extended Characteristics Buffer (Sense Mode)



ZK−5780−GE

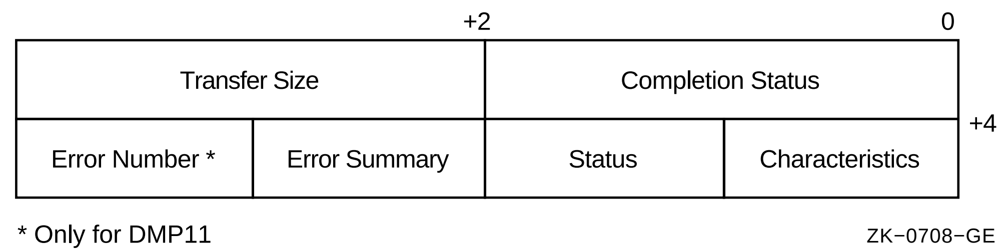
Table 5–9 Tributary Counter Parameter IDs

Parameter ID	Meaning
NMA$C_CTCIR_BRC	Number of bytes received by this station.
NMA$C_CTCIR_BSN	Number of bytes transmitted by this station.
NMA$C_CTCIR_DBR	Number of messages received by this station.
NMA$C_CTCIR_DBS	Number of messages transmitted by this station.
NMA$C_CTCIR_SIE	Number of selection intervals elapsed.
NMA$C_CTCIR_RBE	Remote buffer error bitmap counters. **Value** / **Meaning** 1 — Remote buffer unavailable. 2 — Remote buffer too small.
NMA$C_CTCIR_LBE	Local buffer error bitmap counters. **Value** / **Meaning** 1 — Local buffer unavailable. 2 — Local buffer too small.
NMA$C_CTCIR_SLT	Selection timeout bitmap counters. **Value** / **Meaning** 1 — No attempt to respond was made. 2 — Attempt was made but timeout still occurs.
NMA$C_CTCIR_RRT	Number of SACK settings when REP received.
NMA$C_CTCIR_LRT	Number of SREP settings.
NMA$C_CTCIR_DEI	Data error inbound bitmap counters. **Value** / **Meaning** 1 — NAK transmitted header CRC error. 2 — NAK transmitted data CRC error. 4 — NAK transmitted REP response.
NMA$C_CTCIR_DEO	Data error outbound bitmap counters. **Value** / **Meaning** 1 — NAK received header CRC error. 2 — NAK received data CRC error. 4 — NAK received REP response.

5.5 I/O Status Block

The I/O status block (IOSB) for all asynchronous DDCMP functions is shown in Figure 5–4. Appendix A lists the completion status returns for these functions. (The OpenVMS system messages documentation provides explanations and suggested user actions for these returns.)

Figure 5–4 IOSB Contents for the DDCMP Functions



In addition to the completion status, the first longword of the IOSB returns either the size (in bytes) of the data transfer or the size (in bytes) of the characteristics buffer returned by a sense mode function. The second longword returns the line status bits listed in Table 5–2 and the error summary bits listed in Table 5–3.

A

I/O Function Codes

This appendix lists the function codes and function modifiers defined by the $IODEF macro within the OpenVMS VAX environment. The associated arguments for these functions are also provided.

A.1 DMC11/DMR11 Interface Driver

Functions	Arguments	Modifiers
IO$_READLBLK IO$_READVBLK IO$_READPBLK	P1 - buffer address P2 - message size	IO$M_DSABLMBX IO$M_NOW
IO$_WRITELBLK IO$_WRITEVBLK IO$_WRITEPBLK	P1 - buffer address P2 - message size	IO$M_ENABLMBX[1]
IO$_SETMODE IO$_SETCHAR	P1 - characteristics buffer address	
IO$_SETMODE!IO$M_ATTNAST IO$_SETMODE!IO$M_ATTNAST	P1 - AST service routine address P2 - (ignored) P3 - AST access mode	
IO$_SETMODE!IO$M_SHUTDOWN IO$_SETCHAR!IO$M_SHUTDOWN	P1 - characteristics block address	
IO$_SETMODE!IO$M_STARTUP IO$_SETCHAR!IO$M_STARTUP	P1 - characteristics block address P2 - (ignored) P3 - receive message blocks	

[1]Only for IO$_WRITELBLK and IO$_WRITEPBLK

QIO Status Returns	
SS$_ABORT	SS$_BADPARAM
SS$_DATAOVERUN	SS$_DEVACTIVE
SS$_DEVOFFLINE	SS$_ENDOFFILE
SS$_NORMAL	

A.2 DMP11 and DMF32 Interface Drivers

Functions	Arguments
IO$_READLBLK[!IO$M_NOW] IO$_READVBLK[!IO$M_NOW] IO$_READPBLK[!IO$M_NOW] IO$_WRITELBLK IO$_WRITEVBLK IO$_WRITEPBLK	P1- buffer address P2 - buffer size P6 - diagnostic buffer address (optional)
IO$_SETMODE IO$_SETCHAR IO$_SETMODE!IO$M_CTRL IO$_SETCHAR!IO$M_CTRL IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP IO$_SETCHAR!IO$M_CTRL!IO$M_STARTUP IO$_SETMODE!IO$M_STARTUP IO$_SETCHAR!IO$M_STARTUP IO$_SETMODE!IO$M_SHUTDOWN IO$_SETCHAR!IO$M_SHUTDOWN IO$_SETMODE!IO$M_CTRL!IO$M_SHUTDOWN IO$_SETCHAR!IO$M_CTRL!IO$M_SHUTDOWN	P1 - characteristics buffer address (optional) P2 - extended characteristics buffer descriptor address (optional) P3 - receive message blocks (optional) P6 - diagnostic buffer address (optional)
IO$_SETMODE!IO$M_ATTNAST IO$_SETCHAR!IO$M_ATTNAST	P1 - AST service routine address P2 - (ignored) P3 - access mode to deliver AST
IO$_SETMODE!IO$M_SET_MODEM[1] IO$_SETCHAR!IO$M_SET_MODEM[1] IO$_SENSEMODE!IO$M_RD_MODEM IO$_SENSEMODE!IO$M_CTRL !IO$M_RD_MODEM[1]	P1 - modem status buffer address
IO$_SENSEMODE IO$_SENSEMODE!IO$M_CTRL	P1 - characteristics buffer address (optional) P2 - extended characteristics buffer descriptor address (optional)
IO$_SENSEMODE!IO$M_RD_COUNTS[2] IO$_SENSEMODE!IO$M_CLR_COUNTS[2] IO$_SENSEMODE!IO$M_RD_COUNTS !IO$M_CLR_COUNTS[2] IO$_SENSEMODE!IO$M_CTRL !IO$M_RD_COUNTS[3] IO$_SENSEMODE!IO$M_CTRL !IO$M_CLR_COUNTS[3] IO$_SENSEMODE!IO$M_CTRL !IO$M_RD_COUNTS !IO$M_CLR_COUNTS[3]	P1 - (ignored) P2 - counter buffer descriptor address
IO$_SENSEMODE!IO$M_RD_MEM[1] IO$_SENSEMODE!IO$M_RD_MEM !IO$M_CTRL[1]	P1 - status slot buffer address P2 - tributary status slot address
IO$_CLEAN	(none)

[1]Only for DMP11

[2]Only for DDCMP

[3]Only for DDCMP and LAPB

QIO Status Returns	
SS$_ABORT	SS$_BADPARAM
SS$_BUFFEROVF	SS$_CANCEL
SS$_DEVACTIVE	SS$_DEVICEFULL
SS$_DEVINACT	SS$_DEVOFFLINE
SS$_ENDOFFILE	SS$_NORMAL

A.3 DR11–W/DRV11–WA Interface Driver

Functions	Arguments	Modifiers
IO$_READLBLK IO$_READVBLK IO$_READPBLK IO$_WRITELBLK IO$_WRITEVBLK IO$_WRITEPBLK	P1 - buffer address P2 - buffer size P3 - timeout period P4 - CSR value P5 - ODR value	IO$M_SETFNCT IO$M_WORD[1] IO$M_TIMED IO$M_CYCLE IO$M_RESET
IO$_SETMODE IO$_SETCHAR	P1 - characteristics buffer address P3 - access mode	IO$M_ATTNAST IO$M_DATAPATH[2]

[1]Not applicable to DRV11–WA

[2]Only for IO$_SETCHAR

QIO Status Returns	
SS$_BADPARAM	SS$_CANCEL
SS$_CTRLERR	SS$_DEVACTIVE
SS$_DRVERR	SS$_EXQUOTA
SS$_NOPRIV	SS$_NORMAL
SS$_OPINCOMPL	SS$_PARITY
SS$_TIMEOUT	

A.4 DR32 Interface Driver

Functions	Arguments	Modifiers
IO$_LOADMCODE	P1 - starting address of microcode to be loaded P2 - load byte count	
IO$_STARTDATA	P1 - starting address of data transfer command table P2 - length of the data transfer command table	IO$M_SETEVF

High-Level Language	Function
XF$SETUP	Defines command and buffer areas; initializes queues
XF$STARTDEV	Issues a request that starts the DR32
XF$FREESET	Releases command packets onto FREEQ
XF$PKTBLD	Builds command packets; releases them onto INPTQ
XF$GETPKT	Removes a command packet from TERMQ
XF$CLEANUP	Deassigns the device channel and deallocates the command area

QIO Status Returns	
SS$_ABORT	SS$_BADPARAM
SS$_BADQUEUEHDR	SS$_BUFNOTALIGN
SS$_CANCEL	SS$_CTRLERR
SS$_DEVACTIVE	SS$_DEVREQERR
SS$_EXQUOTA	SS$_INSFMEM
SS$_IVBUFLEN	SS$_MCNOTVALID
SS$_NORMAL	SS$_PARITY
SS$_POWERFAIL	

A.5 Asynchronous DDCMP DUP11 Interface Driver

Functions	Arguments
IO$_READLBLK[!IO$M_NOW] IO$_READVBLK[!IO$M_NOW] IO$_READPBLK[!IO$M_NOW] IO$_WRITELBLK IO$_WRITEVBLK IO$_WRITEPBLK	P1 - buffer address P2 - buffer size
IO$_SETMODE IO$_SETCHAR IO$_SETMODE!IO$M_STARTUP IO$_SETCHAR!IO$M_STARTUP IO$_SETMODE!IO$M_CTRL IO$_SETCHAR!IO$M_CTRL IO$_SETMODE!IO$M_CTRL!IO$M_STARTUP IO$_SETCHAR!IO$M_CTRL!IO$M_STARTUP IO$_SETMODE!IO$M_SHUTDOWN IO$_SETCHAR!IO$M_SHUTDOWN IO$_SETMODE!IO$M_CTRL!IO$M_SHUTDOWN IO$_SETCHAR!IO$M_CTRL!IO$M_SHUTDOWN	P2 - buffer descriptor address (optional)

Functions	Arguments
IO$_SETMODE!IO$M_ATTNAST IO$_SETCHAR!IO$M_ATTNAST	P1 - AST service routine address P2 - (ignored) P3 - access mode to deliver AST
IO$_SENSEMODE IO$_SENSEMODE!IO$M_CTRL IO$_SENSEMODE!IO$M_RD_COUNTS IO$_SENSEMODE!IO$M_CLR_COUNTS IO$_SENSEMODE!IO$M_RD_COUNTS !IO$M_CLR_COUNTS IO$_SENSEMODE!IO$M_CTRL !IO$M_RD_COUNTS IO$_SENSEMODE!IO$M_CTRL !IO$M_CLR_COUNTS	P1 - (ignored) P2 - buffer descriptor address

QIO Status Returns	
SS$_ABORT	SS$_BADPARAM
SS$_BUFFEROVF	SS$_CANCEL
SS$_DEVACTIVE	SS$_DEVICEFULL
SS$_DEVINACT	SS$_DEVOFFLINE
SS$_ENDOFFILE	SS$_NORMAL

Index

A

C

E

F

I

M

O

P

Q

S

X