```
GGGGGGGGGG    555555555555555555555555555555555555555555555555555555555555    GGGGGGGGGG
GGGGGGGGGG  Artificial Intelligence Technology Center - LM02 Marlboro Mass.   GGGGGGGGGG
GGGGGGGGGG    555555555555555555555555555555555555555555555555555555555555    GGGGGGGGGG


                           EEEEE  V   V   AAA   N   N   SSSS
                           E      V   V  A   A  N   N  S
                           E      V   V  A   A  NN  N  S
                           EEEE   V   V  A   A  N N N   SSS
                           E      V   V  AAAAA  N  NN      S
                           E       V V   A   A  N   N      S
                           EEEEE    V    A   A  N   N  SSSS


                           XX      XX    QQQQQQ    PPPPPPPP
                           XX      XX    QQQQQQ    PPPPPPPP
                           XX      XX  QQ      QQ  PP      PP
                           XX      XX  QQ      QQ  PP      PP
                             XX  XX    QQ      QQ  PP      PP
                             XX  XX    QQ      QQ  PP      PP
                               XX      QQ      QQ  PPPPPPPP
                               XX      QQ      QQ  PPPPPPPP
                             XX  XX    QQ  QQ  QQ  PP
                             XX  XX    QQ  QQ  QQ  PP
                           XX      XX  QQ    QQ    PP
                           XX      XX  QQ    QQ    PP
                           XX      XX    QQQQ  QQ  PP
                           XX      XX    QQQQ  QQ  PP


                     LL          NN      NN    333333        ;;;;      222222
                     LL          NN      NN    333333        ;;;;      222222
                     LL          NN      NN  33      33      ;;;;    22      22
                     LL          NN      NN  33      33      ;;;;    22      22
                     LL          NNNN    NN          33                    22
                     LL          NNNN    NN          33                    22
                     LL          NN  NN  NN        33      ;;;;          22
                     LL          NN  NN  NN        33      ;;;;          22
                     LL          NN    NNNN          33      ;;;;        22
                     LL          NN    NNNN          33      ;;;;        22
             ....    LL          NN      NN  33      33        ;;      22
             ....    LL          NN      NN  33      33        ;;      22
             ....    LLLLLLLLLL  NN      NN    333333        ;;      2222222222
             ....    LLLLLLLLLL  NN      NN    333333        ;;      2222222222


File _QED$DUA0:[SYSCOMMON.DQS$SERVER]XQP.LN3;2 (172,59,0), last revised on  7-MAY-1992
16:11, is a 665 block sequential file owned by UIC [DQS$SERVER].  The records are
variable length with implied (CR) carriage control.  The longest record is 0 bytes.

Job XQP (891) queued to LN3_D4 on 7-MAY-1992 16:11 by user EVANS, UIC [DQS$SERVER],
under account LRNLAB:: at priority 100, started on printer _QED$TXA4: on 7-MAY-1992
16:17 from queue LN3_D4.

GGGGGGGGGG    555555555555555555555555555555555555555555555555555555555555    GGGGGGGGGG
GGGGGGGGGG      Digital Equipment Corporation - VAX/VMS Version V5.5-FB5      GGGGGGGGGG
GGGGGGGGGG    555555555555555555555555555555555555555555555555555555555555    GGGGGGGGGG
```

Contents

Figures

Tables

Chapter 6
The XQP and I/O Processing

I/O processing is the handling of a user request for an input/output operation to the driver associated with a particular device. I/O processing can be divided into three phases:

- I/O request preprocessing
- Driver-specific processing
- I/O postprocessing

I/O request preprocessing is handled in the VMS executive by the $QIO system service. Driver-specific processing is performed by the driver associated with a particular device. I/O postprocessing is also handled by other VMS executive routines.

Although I/O can complete without involving the file system, a specific part of the file system called the **extended QIO processor** (XQP) must intervene to perform additional processing that cannot be done by either the QIO system service or by the driver. Specifically, the XQP performs the following tasks:

- Processing a non-transfer request (for example, a file access).
- Handling bad blocks found in the course of performing an I/O operation.
- Processing a transfer request when the current information in memory is insufficient to convert the virtual blocks of a file to the logical blocks of the disk.

This chapter describes I/O pre- and postprocessing, which is essentially the flow of I/O requests prior to and beyond the XQP. The following topics are discussed:

- How and where the XQP is mapped
- The layout of impure storage
- The $QIO system service interface to the XQP
- The format of I/O request packets
- FDT action routines

- XQP packet building and processing
- XQP kernel stack switching
- Error handling
- Posting I/O status to the user

6.1 XQP Initialization

The Files-11 image (F11BXQP.EXE) contains only pure code, which is code that is never written to and thus cannot be modified. It is mapped into P1, or process control, space when the process is created. The mapping can be performed quickly and efficiently because no I/O needs to be done for the process at this time. The XQPMERGE routine in the SYS facility module PROCSTRT performs the mapping operation. Because it is kernel mode code, this routine is optimized. A single permanent global section is created for the F11BXQP image during system initialization by the SYSINIT process.

If the system parameter ACP_XQP_RES is set, SYSINIT maps the code into physical memory so that global valid page faults may be avoided. However, under exceptional circumstances, the ACP_XQP_RES parameter may not be set (for example, on a system with restricted memory that shows little file activity or a system with a small number of users), the code is not resident.

6.1.1 Allocating Impure Storage

Once the code has been mapped, the XQPMERGE routine jumps to the lowest address mapped—the initialization routine is the INITXQP in the module DISPATCH. This routine is linked as the first in the image.

The initialization routine INITXQP changes mode to kernel, specifying the INIT_FCP routine in the INIFCP module. This routine calls the $EXPREG system service to add virtual pages in P1 space to map the impure storage area. It also sets the process cell CTL$GL_F11BXQP to point to the queue header F11B$Q_XQPQUEUE (or XQP_QUEUE) in the XQP impure area.

There are three major portions of the XQP impure area:

- A private per-process kernel stack for use by the XQP
- An XQP queue
- Per-process XQP data, which includes a context save area

The INIT_FCP routine locks into the working set of the process the area for the kernel stack, the data of the impure area, and the code, which are referenced at elevated IPL (any IPL greater than 2); that is, the pages of the impure area are counted as part of the working set size. The routine also assigns a channel for the XQP and initializes the XQP queue header.

At the top of the XQP impure area is the XQP private stack. The stack occupies 5 pages. When the XQP dispatcher processes requests, the process uses this private kernel stack instead of the normal kernel stack. The stack thus contains normal call frames and data normally placed on the kernel stack. How the XQP switches from one stack to the other is discussed in more detail in Section 6.4.3.

Figure 6-1 shows the F11BXQP structure, which is part of the XQP impure area. It is pointed to by the process cell CTL$GL_F11BXQP. The F11BXQP structure is an external, global structure that defines per-process XQP symbols. It overlays the top portion of the actual per-process XQP symbols defined by FCPDEF.B32 in the F11X facility. The symbols defined by FCPDEF are internal to the XQP, but the F11BXQP structure allows the symbols defining the size and location of the XQP to be visible to the System Dump Analyzer Utility (SDA).

Figure 6-1 Format of the F11BXQP Structure

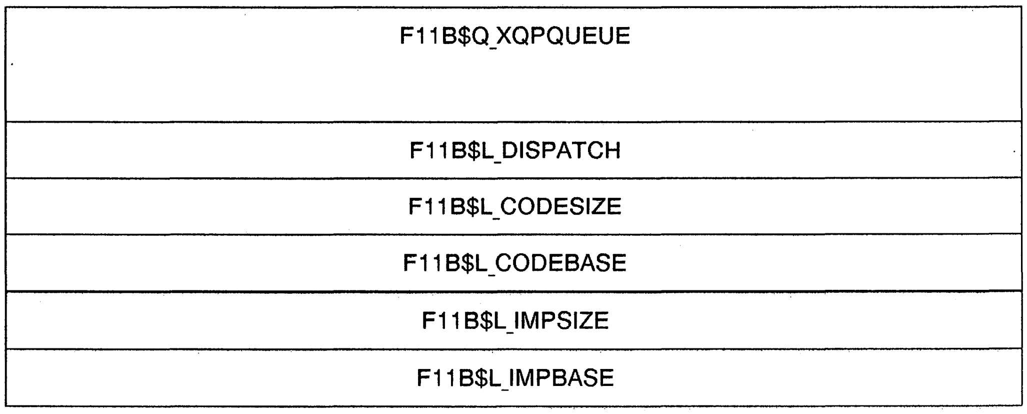

Table 6-1 Contents of the F11BXQP Structure

Field Name	Description
F11B$Q_XQPQUEUE	XQP per-process queue header. This queue contains the I/O request packets (IRPs) that are currently queued to the XQP by the process. Each IRP describes an individual I/O request.
F11B$L_DISPATCH	Address of XQP dispatch routine EXE$QXQPPKT in the module SYSQIOREQ. This routine represents the first level of I/O dispatching. F11B$L_DISPATCH is a pointer to the DISPATCH routine in the DISPATCH module.
F11B$L_CODESIZE	Size of XQP code in bytes.

Table 6-1 (Cont.) Contents of the F11BXQP Structure

Field Name	Description
F11B$L_CODEBASE	Base address of XQP code. This field contains the starting address of the pure XQP code in P1 space.
F11B$L_IMPSIZE	Size of impure area in bytes.
F11B$L_IMPBASE	Base address of XQP impure area. This field dcontains the starting address (that is, the top of the XQP private kernel stack) of the XQP impure data storage area in P1 space.

The space for the XQP impure area is allocated dynamically, and it can be allocated anywhere in P1 space because it is based off a single register. Register R10 is the base register for the XQP impure area, and it is initialized to the address in CONTEXT_START.

Figure 6-2 shows the layout of the XQP impure area and code in the process control region. The shaded area pointed to by the process cell CTL$GL_F11BXQP is expanded in Figure 6-3.

Figure 6-2 Layout of the XQP

```
F11B$L_IMPBASE --> +-----------------------------+ \
                   |                             |  \
                   |      XQP internal stack     |   \ XQP impure area
                   |                             |   / F11B$L_IMPSIZE =
CTL$GL_F11BXQP --> +-----------------------------+  /  length
                   |\\\\\\\\\\\\\\\\\\\\\\\\\\\\\|
F11B$L_CODEBASE -> +-----------------------------+
                   |                             | \
                   |                             |  \
                   |          XQP code           |   \ F11B$L_CODESIZE
                   |                             |   /
                   |                             |  /
                   +-----------------------------+ /
```

Figure 6-3 shows a further expansion of the XQP impure area. The impure storage area is delimited by the symbols STORAGE_START and STORAGE_END. The symbol L_DATA_START also points to the beginning of this area.

The pages represented by the cells located between L_DATA_START and L_DATA_END are locked into the working set of the process because they must be present at elevated IPL.

IMPURE_START and IMPURE_END delimit the cells that are initialized to known values (usually 0) by the per-request initialization routine.

CONTEXT_START and CONTEXT_END mark the beginning and end of the reenterable context area, which must be saved when a secondary operation is performed.

The context save area, delimited by CONTEXT_SAVE and CONTEXT_SAVE_END, is the area in which the primary context is saved when a secondary operation is performed. This topic is covered in more detail in Section 6.4.2.

Figure 6-3 Format of the Impure Area

```
STORAGE_START -->
L_DATA_START ---> +---------------------------+
                  |        XQP_STACK          |
                  |        (5 pages)          |
                  |                           |
CTL$GL_F11BXQP -> +---------------------------+\
                  |        XQP_QUEUE          | \
                  +--                       --+  \
                  |                           |   \
                  +---------------------------+    \
                  |        XQP_DISPATCHER     |     \
                  +---------------------------+      \   Overlaid by
                  |        CODE_SIZE          |        -- system-wide
                  +---------------------------+      / F11BXQP structure
                  |        CODE_ADDRESS       |     /
                  +---------------------------+    /
                  |        DATA_SIZE          |   /
                  +---------------------------+  /
                  |        DATA_ADDRESS       | /
                  +---------------------------+/
                  |             :             |
                  +---------------------------+
                  |        BLOCK_LOCKID       |
IMPURE_START ---> +---------------------------+
                  |        USER_STATUS        |
                  +--                       --+
                  |                           |
                  +---------------------------+
                  |             :             |
                  +---------------------------+
                  |        CACHE_HDR          |
CONTEXT_START --> +---------------------------+
                  |        CLEANUP_FLAGS      |
                  +---------------------------+
                  |             :             |
                  +---------------------------+
                  |        PREV_LINK          |
CONTEXT_END   --> +---------------------------+
CONTEXT_SAVE ---> |        Context            |
                  |         save              |
                  |         area              |
CONTEXT_SAVE_END> +---------------------------+
```

(Continued on next page)

Figure 6-3 (Cont.) Format of the Impure Area

```
                  +---------------------------+
                  |        LB_LOCKID          |
                  +---------------------------+
                  |            :              |
                  +---------------------------+
                  |        SECOND_FIB         |
                  +---------------------------+
                  |        LOCAL_ARB          |
L_DATA_END  ----> +---------------------------+
                  |            :              |
                  +---------------------------+
                  |        AUDIT_COUNT        |
IMPURE_END -----> +---------------------------+
                  |        PMS statistics     |
                  |            :              |
                  +---------------------------+
                  |            :              |
                  +---------------------------+
                  |        AUDIT_ARGLIST      |
STORAGE_END --->  +---------------------------+
```

Table 6-2 lists all cells of the XQP impure area, their size, and a short description of each.

Table 6-2 Contents of the XQP Impure Area

Impure Symbol	Size	Description
STORAGE_START	0	Label marking the beginning of the impure storage area.
L_DATA_START	0	Label marking the beginning of data that has been "locked down," or locked in the working set of the process.
XQP_STACK	5 pages	XQP kernel stack.
XQP_QUEUE	2 longwords	Two-longword XQP queue head. This cell corresponds to the F11B$Q_XQPQUEUE field.
XQP_DISPATCHER	Longword	Address of the XQP dispatch routine. This cell corresponds to the F11B$L_DISPATCH field.
CODE_SIZE	Longword	Length of the XQP code. This cell corresponds to the F11B$L_CODESIZE field.

Table 6-2 (Cont.) **Contents of the XQP Impure Area**

Impure Symbol	Size	Description
CODE_ADDRESS	Longword	Base address of the XQP code. This cell corresponds to the F11B$L_CODEBASE field.
DATA_SIZE	Longword	Length of the impure data area. This cell corresponds to the F11B$L_IMPSIZE field.
DATA_ADDRESS	Longword	Base address of the impure data area. This cell corresponds to the F11B$L_IMPBASE field.
PREV_FP	Longword	Saved frame pointer.
PREV_STKLIM	2 longwords	Two-longword saved kernel stack limits.
XQP_STKLIM	2 longwords	Two-longword XQP kernel stack limits.
XQP_SAVFP	Longword	Saved XQP frame pointer.
IO_CCB	Longword	Address of the channel control block of IO_CHANNEL, created by INIT_FCP. This cell is set to CURRENT_UCB by GET_REQUEST and to the new UCB by SWITCH_VOLUME. It is used to refer to the desired UCB by WRITE_BLOCK because buffer write operations because of LRU replacement may be to other than the current UCB.
IO_CHANNEL	Longword	Channel number for I/O. This cell is used to force mount verification on shadow sets; to issue an Unload /Available function when the volume is dismounted; to erase blocks of the index file when the end-of-file is extended; to read and write random blocks; and to erase blocks for highwater and erase-on-return processing.
BLOCK_LOCKID	Longword	Lock ID of the activity-blocking lock held by this process. See Chapter 8 for more information.
IMPURE_START	0	Label marking the start of the impure data area, the cells of which are initialized to known values by the per-request initialization routine.

Table 6-2 (Cont.) Contents of the XQP Impure Area

Impure Symbol	Size	Description
USER_STATUS	2 longwords	I/O status to be returned to user. It is a two-longword vector returned through IRP$L_MEDIA, which forms the I/O status block (IOSB). EXTEND sets the second longword to the size extended, and EXTEND_INDEX purposely zeros it. For a contiguous Extend operation (ALLOC_BITMAP), this value is the largest contiguous extent size found. For a Truncate operation, this value is the number of blocks left in the file such that the truncated file still has an integral number of clusters. READ_WRITEVB sets this value to the second word of the I/O status block returned by the I/O. See Section 6.5.2 for more information.
IO_STATUS	2 longwords	Status block for XQP I/O.
IO_PACKET	Longword	Address of the current I/O request packet, set in the DISPATCHER routine. If this cell contains a value of 0, the XQP is currently idle.
CURRENT_UCB	Longword	Address of the UCB of the current request, set in GET_REQUEST and SWITCH_VOLUME.
CURRENT_VCB	Longword	Address of the VCB of the current request, set in GET_REQUEST and SWITCH_VOLUME.
CURRENT_RVT	Longword	Address of the RVT of the current volume set, or UCB, set in GET_REQUEST.
CURRENT_RVN	Longword	Address of the RVN of the current volume, set in GET_REQUEST and SWITCH_VOLUME.
SAVE_VC_FLAGS	Word	Save volume context flags. These flag bits belong to the allocation lock value block. They contain the quota file buffer sequence number in bits 1 to 15.

Table 6-2 (Cont.) Contents of the XQP Impure Area

Impure Symbol	Size	Description
STSFLGS	Byte	Various internal status flags. These are global flags that allow special processing to be requested by a routine without having to pass extra arguments to the routine.
BLOCK_CHECK	Byte	Operation blocking check.
NEW_FID	Longword	File number of the unrecorded file ID.
NEW_FID_RVN	Longword	RVN of NEW_FID.
HEADER_LBN	Longword	LBN of the last file header read. This value is placed into FCB$L_HDLBN by FILL_FCB.
BITMAP_VBN	Longword	VBN of the current storage map block. This value is used along with BITMAP_RVN to determine the validity of BITMAP_BUFFER. This value is cleared when the allocation lock is released because a bitmap buffer cannot be active at this time. Invalidating the BITMAP_BUFFER will also clear this value.
BITMAP_RVN	Longword	RVN of the current storage map block, BITMAP_BUFFER.
BITMAP_BUFFER	Longword	Address of the current storage map block. This value is used as an optimization in ALLOC_BLOCKS to decide if a storage map block needs to be read. The validity of BITMAP_BUFFER is indicated by a non-zero value in BITMAP_VBN.
SAVE_STATUS	Longword	Saved status. During a Create operation, this cell holds the saved status while attributes are copied in READ_IDX_HEADER. During a Delete operation, it is used to restore the old USER_STATUS if the operation fails.
PRIVS_USED	Quadword	Privileges used to gain access. This bit array is maintained by CHECK_PROTECT. This value can be returned as a read attribute.

Table 6-2 (Cont.) Contents of the XQP Impure Area

Impure Symbol	Size	Description
ACB_ADDR	Longword	Address of the AST control block (ACB) for cross-process ASTs, set in READ_BLOCK to the CDRP portion of the IRP indicated in IO_PACKET.
BFR_LIST	4 quadwords	Listheads for in-process buffers. See Chapter 5 for more information.
BFR_CREDITS	4 words	Buffers credited to the process. See Chapter 5 for more information.
BFRS_USED	4 words	Buffers actually in-process. See Chapter 5 for more information.
CACHE_HDR	Longword	Address of the buffer cache header, set by GET_REQD_BFR_CREDITS.
CONTEXT_START	0	Label marking the beginning of the reenterable context area, which must be saved when a secondary operation is performed.
CLEANUP_FLAGS	Longword	Cleanup action flags.
FILE_HEADER	Longword	Address of current file header, set by CREATE and CREATE_HEADER. EXTEND_HEADER sets this value to the new extension header. DELETE_FILE zeros FILE_HEADER when the new header is written.
PRIMARY_FCB	Longword	Address of primary file FCB. This cell is set by following routines: GET_REQUEST, ACCESS, CREATE, MARK_DELETE, EXTEND_CONTIG, EXTEND_INDEX, OPEN_FILE, MODIFY, DEACC_QFILE, CONN_QFILE, and SHUFFLE_DIR. It is cleared by CLOSE_FILE and by MARK_DELETE when the file is deleted. It is also cleared by GET_FIB, ACCESS, and MODIFY when the FID in the user's FIB does not match that of the FCB associated with the channel.

Table 6-2 (Cont.) Contents of the XQP Impure Area

Impure Symbol	Size	Description
CURRENT_WINDOW	Longword	Address of the file window. This cell is set by the following routines: GET_REQUEST, ACCESS, CREATE, EXTEND_INDEX, and OPEN_FILE. It is cleared by GET_FIB, ACCESS, DELETE, and MODIFY when the FID in the user's FIB does not match that of the FCB associated with the channel.
CURRENT_FIB	Longword	Pointer to FIB currently in use, set to LOCAL_FIB by GET_FIB and GET_REQUEST. It is set to SECOND_FIB by SAVE_CONTEXT (LOCAL_FIB is not in the context save area).
CURR_LCKINDX	Longword	Current file header lock index. Refer to Chapter 7 for more information.
PRIM_LCKINDX	Longword	Primary file lock basis index. Refer to Chapter 7 for more information.
LOC_RVN	Longword	RVN specified by placement data, set by GET_LOC.
LOC_LBN	Longword	LBN specified by placement data, set by GET_LOC.
UNREC_LBN	Longword	Starting LBN of unrecorded blocks.
UNREC_COUNT	Longword	Count of unrecorded blocks.
UNREC_RVN	Longword	RVN containing unrecorded blocks.
PREV_LINK	6 bytes	Old back link of file. This length of this cell is specified by the FID$C_LENGTH constant, which is currently 6 bytes.
CONTEXT_END	0	Label marking the end of secondary context.
CONTEXT_SAVE	54 bytes	Size of the context save area, which is the area in which the primary context is saved when a secondary operation is performed.
CONTEXT_SAVE_END	0	Label marking the end of the context save area.

Table 6-2 (Cont.) Contents of the XQP Impure Area

Impure Symbol	Size	Description
LB_LOCKID	5 longwords	Serial lock IDs. This length of this cell is determined by the LB_NUM literal (representing the number of serial lock blocks), which is currently 5. For more information, see Chapter 7.
LB_BASIS	5 longwords	Lock name bases. This cell contains the system addresses of lock names.
LB_HDRSEQ	5 longwords	File header cache sequence numbers. For more information, see Chapter 5.
LB_DATASEQ	5 longwords	File data block cache sequence number. For more information, see Chapter 5.
LB_FILESIZE	5 longwords	Value block file size. This value is taken from a longword in the lock value block containing the size of the file so that another process can open the file from another node, thus allowing shared access across a VAXcluster.
DIR_FCB	Longword	FCB of directory file, set in DIR_ACCESS. This field is cleared in DELETE if the directory itself is deleted.
DIR_LCKINDX	Longword	Directory lock basis index. For more information, see Chapter 7.
DIR_RECORD	Longword	Record number of found directory entry within the block. This value is maintained by DIR_SCAN and FIND. It is zeroed before an Enter operation. The value in DIR_RECORD, plus 1, becomes the low order 6 bits of the wildcard context (FIB$L_WCC) returned to the user.
DIR_CONTEXT	112 bytes	Current directory context. The directory context is saved within ENTER when it is necessary to do another DIR_SCAN to find the lowest entry to remove. It is restored by RESTORE_DIR when a directory operation is done at cleanup time.
OLD_VERSION_FID	6 bytes	FID of the previous version of the file, set by DIR_SCAN.
PREV_VERSION	Longword	Version number of previous directory entry.

Table 6-2 (Cont.) Contents of the XQP Impure Area

Impure Symbol	Size	Description
PREV_NAME	80+1 bytes	Name of the previous entry.
PADDING_0	1 byte	Alignment byte.
PREV_INAME	86 bytes	Previous internal file name from the file header. It is used during a Rename function.
SUPER_FID	6 bytes	File ID of the superseded file.
LOCAL_FIB	64 longwords	Primary FIB of this operation (see CURRENT_FIB). The length of this cell is determined by the constant FIB$C_LENGTH.
SECOND_FIB	64 longwords	FIB for a secondary file operation (see CURRENT_FIB). The length of this cell is specified by the constant FIB$C_LENGTH.
LOCAL_ARB	?	Local copy of the caller's access rights block (ARB).
L_DATA_END	0	Label marking the end of the data that has been locked into the working set of the process.
QUOTA_RECORD	Longword	Record number of the quota file entry, returned as wildcard context to the user.
FREE_QUOTA	Longword	Record number of the free quota file entry.
REAL_Q_REC	Longword	Buffer address of the quota record read.
QUOTA_INDEX	Longword	Cache index of the quota cache entry found.
DUMMY_REC	?	Dummy quota record for cache contents. This cell is a special case in WRITE_QUOTA, meaning that the quota record pointer does not point into a cache buffer.
AUDIT_COUNT	Longword	Number of argument lists in AUDIT_ARGLIST.
IMPURE_END	0	Label marking the end of the impure data area.

Table 6-2 (Cont.) Contents of the XQP Impure Area

Impure Symbol	Size	Description
MATCHING_ACE	?	Matching access control entry (ACE) storage, set by CHECK_PROTECT to the ACE which the access check matched, returnable via READ_ATTRIB.
FILE_SPEC_LEN	Word	Full file specification length.
FULL_FILE_SPEC	1022 bytes	Full file specification storage, including full directory specification.[1] This cell is a storage area to hold the output of FID_TO_SPEC, used by WRITE_AUDIT and READ_ATTRIB.
PMS_TOT_READ	Longword	Total number of disk reads.
PMS_TOT_WRITE	Longword	Total number of disk writes.
PMS_TOT_CACHE	Longword	Total number of cache reads.
PMS_FNC_READ	Longword	Total number of Read functions.
PMS_FNC_WRITE	Longword	Total number of Write functions.
PMS_FNC_CACHE	Longword	Total number of cache hits, or how many times the desired record was in the cache.
PMS_FNC_CPU	Longword	Total CPU time used per function.
PMS_FNC_PFA	Longword	Total number of page faults incurred.
PMS_SUB_NEST	Longword	Nested subfunction flag.
PMS_SUB_FUNC	Longword	Subfunction code.
PMS_SUB_READ	Longword	Number of subfunction read operations.
PMS_SUB_WRITE	Longword	Number of subfunction write operations.
PMS_SUB_CACHE	Longword	Number of subfunction cache hits.
PMS_SUB_CPU	Longword	Subfunction CPU time used.
PMS_SUB_PFA	Longword	Number of subfunction page faults.
AUDIT_ARGLIST	64 bytes	Security auditing argument lists. This cell is used to accumulate audit records.
STORAGE_END	0	Label marking the end of the impure storage area.

[1]In the absence of concealed device definition.

In addition, the SYSINIT process, the first process to be merged in this way when the system is booted, creates a permanent mailbox (ACP$BADBLOCK_MBX, channel MBX_CHAN) to communicate with the bad block scanner.

6.2 The XQP Call Interface

The user interface to the XQP is provided by the Queue I/O Request ($QIO) system service. All file system functions are QIOs. When a user process issues an I/O request, QIO gains control and coordinates the preprocessing of the request. The QIO system service is dispatched by a system service vector in P1 space, which changes the access mode of the process to kernel and dispatches to the EXE$QIO procedure.

Also used by the XQP are the I/O request packets (IRPs), the function decision table (FDT) of the pertinent driver, and the driver dispatch table (DDT).

6.2.1 The I/O Request Packet

The **I/O request packet** (IRP) is the basic argument block passed to the file system for all functions. An IRP is a piece of nonpaged pool that describes the I/O request. When a process requests that I/O be performed, an IRP is constructed in a standard format.

The IRP contains fields into which the system I/O preprocessing routines write information. The packet also includes buffer addresses, a pointer to the target device, I/O function codes, and pointers to the I/O database.

Some of the packet is device-independent information filled in by the $QIO system service; the rest is device-dependent information filled in by function decision table routines. The IRP is first processed by the file system FDT routines, which later queue the IRP to the XQP, if necessary.

IRPs are a part of the I/O database. They are allocated in system space (nonpaged pool) so the user cannot change the parameters after the $QIO system service validates them and so the driver can access them when the user process is no longer resides in memory. QIO fills in the first part of the packet from the device-independent parameters, of which there are six:

1. Event flag number (EFN)
2. Channel number
3. I/O function code
4. AST parameter
5. AST routine address
6. I/O status block (IOSB) address

The fields of the IRP are shown in Figure 6-4 and are described in Table 6-3. Note that the fields of the figure run right to left.

Figure 6-4 Format of the I/O Request Packet

IRP$L_IOQFL		
IRP$L_IOQBL		
IRP$B_RMOD	IRP$B_TYPE	IRP$W_SIZE
IRP$L_PID		
IRP$L_AST		
IRP$L_ASTPRM		
IRP$L_WIND		
IRP$L_UCB		
IRP$B_PRI	IRP$B_EFN	IRP$W_FUNC
IRP$L_IOSB		
IRP$W_STS		IRP$W_CHAN
IRP$L_SVAPTE		
IRP$L_BCNT		IRP$W_BOFF
reserved		IRP$L_BCNT
IRP$L_MEDIA		
IRP$B_CARCON		
IRP$L_ABCNT		
IRP$L_OBCNT		
IRP$L_SEGVBN		
IRP$L_DIAGBUF		
IRP$L_SEQNUM		
IRP$L_EXTEND		
IRP$L_ARB		
IRP$L_KEYDESC		

Table 6-3 Contents of the I/O Request Packet

Field Name	Description
IRP$L_IOQFL	I/O queue forward link. This field contains the address of the listhead of the queue for all system-wide pending I/O.
IRP$L_IOQBL	I/O queue backward link.
IRP$W_SIZE	Size of the IRP in bytes. The EXE$QIO routine writes the constant IRP$C_LENGTH into this field when the routine allocates and fills an IRP.
IRP$B_TYPE	Structure type for an IRP. The EXE$QIO routine writes the constnat DYN$C_IRP into this field when the routine allocates and fills an IRP.
IRP$B_RMOD	Access mode of request. The EXE$QIO routine obtains the processor access mode from the PSL and writes the value into this field. The following field is defined within IRP$B_RMOD. IRP$V_MODE Mode subfield. This field indicates the mode of AST delivery resulting from the completion of the QIO. In this case, for efficiency, the front part of the IRP has been allocated as an ACB. This field is 2 bits long, and occupies bit positions 24 and 25.
IRP$L_PID	Process ID of requesting process. The EXE$QIO routine obtains the PID of the process that issued the I/O request from the PCB and writes the value into this field.
IRP$L_AST	Address of AST routine. If the process specified an AST routine address in the call to the $QIO system service, EXE$QIO writes the address in this field.
IRP$L_ASTPRM	AST parameter. If the process specified an AST routine and a parameter to that AST routine in the $QIO call, EXE$QIO writes the parameter in this field.
IRP$L_WIND	Address of the window control block. This field contains the address of the WCB that describes the file being accessed in the I/O request.
IRP$L_UCB	Address of the device UCB. The EXE$QIO routine copies the address of the UCB for the device assigned to the process I/O channel into this field.
IRP$W_FUNC	I/O function code and modifiers. This field specifies the I/O function code that identifies the function to be performed for the I/O request. The EXD$QIO routine and driver FDT routines map the code value to its most basic level and copy the reduced value into this field.

Table 6-3 (Cont.) Contents of the I/O Request Packet

Field Name	Description
	Based on this function code, EXE$QIO calls FDT action routines to preprocess the I/O request. Six bits of the function code describe the basic function, and the remaining 10 bits modify the function. The following fields are defined within IRP$W_FUNC. IRP$V_FCODE — Function code field. This field is 6 bits long, and starts at bit 0. IRP$V_FMOD — Function modifier field. This field is 10 bits long, and starts at bit 6.
IRP$B_EFN	Event flag number and event group. If the I/O request call does not specify an event flag number, EXE$QIO uses event flag 0.
IRP$B_PRI	Base priority of the requesting process. EXE$QIO obtains a value for this field from the PCB. This field is used when an IRP is inserted into a priority-ordered pending I/O queue.
IRP$L_IOSB	Address of the I/O status block. This field receives the final status of the I/O request at I/O completion. EXE$QIO writes a value into this field if the I/O request call specifies an IOSB address.
IRP$W_CHAN	Process I/O channel number. This field contains the index number of the process I/O channel for the I/O request.
IRP$W_STS	Request status. EXE$QIO and FDT routines modify this field according to the current status of the I/O request. I/O postprocessing routines read this field to determine what postprocessing is necessary. The status word is used to identify whether the I/O is direct I/O or buffered I/O. **Direct I/O** is performed by locking the pages of the user buffer in physical memory. **Buffered I/O**, on the other hand, is performed by writing the data to a user buffer in nonpaged pool with a special kernel-mode AST. This field also contains bits to specify pager I/O and swapper I/O, which are performed by special system subroutines, not by the $QIO system service.

Table 6-3 (Cont.) Contents of the I/O Request Packet

Field Name	Description
	There is also a bit that specifies a virtual request (a request for file I/O). If a virtual request for file I/O completes with an error caused by a bad disk block, the XQP is informed as part of bad block support. The XQP then records in the file's header that a bad block was found, so that when the file is deleted, appropriate action can be taken.
	Other bits in this field specify **complex buffered I/O**, **chained buffered I/O**, and **Long virtual I/O**.
	Complex buffered I/O and chained complex buffered I/O are used by the XQP. Complex buffered I/O is used for Access and Deacess ACP functions, and chained complex buffered I/O is used by the NETACP for transmit QIO requests.
	Long virtual I/O is virtually contiguous in the file, but physically discontiguous on the disk. This type of I/O is usually done by the VMS executive.
	The following bits are defined within IRP$W_STS. These bits are adjacent and in order.

IRP$V_BUFIO	Buffered I/O function. This is bit 16.
IRP$V_FUNC	Function bit. A set bit indicates a read function; a clear bit indicates a write function. This is bit 17.
IRP$V_PAGIO	Paging I/O function. This is bit 18.
IRP$V_COMPLX	Complex buffered I/O function. This is bit 19.
IRP$V_VIRTUAL	Virtual I/O function. This is bit 20.
IRP$V_CHAINED	Chained buffered I/O function. This is bit 21.
IRP$V_SWAPIO	Swap I/O function. This is bit 22.
IRP$V_DIAGBUF	Diagnostic buffer allocated. This is bit 23.
IRP$V_PHYSIO	Physical I/O function. This is bit 24.

Table 6-3 (Cont.) Contents of the I/O Request Packet

Field Name	Description	
	IRP$V_TERMIO	Terminal I/O function. This is bit 25.
	IRP$V_MBXIO	Mailbox buffered read function. This is bit 26.
	IRP$V_EXTEND	An extended IRP (an IRPE) is linked to this IRP. This is bit 27.
	IRP$V_FILACP	File ACP I/O (both DIOCNT and BIOCNT). This is bit 28.
	IRP$V_MVIRP	Mount verification IRP function. This is bit 29.
	IRP$V_JNL_REMREQ	Remote I/O (slave) request. This is bit 30.
	IRP$V_KEY	IRP$L_KEYDESC contains the address of a key for used for encryption. This is bit 31.
IRP$L_SVAPTE	This field has two functions. For a **direct I/O** transfer, this field contains the system virtual address of first page table entry (PTE) of the I/O transfer buffer. For a **buffered I/O** transfer, this field contains the address of the buffer in system address space.	
IRP$W_BOFF	Byte offset in first page of a direct I/O transfer. FDT routines calculate this offset and write the field. For buffered I/O transfers, FDT routines must write the number of bytes to be charged to the process in this field because these bytes are used for a system buffer. I/O postprocessing uses this field with the IRP$L_BCNT and IRP$L_SVAPTE fields to unlock pages locked for direct I/O. For buffered I/O, I/O postprocessing adds the value in this field to the process byte count quota.	
IRP$L_BCNT	Byte count of transfer. This field contains the the count value, which is calculated by FDT routines. For a buffered I/O read function, I/O postprocessing uses IRP$L_BCNT to determine how many bytes of data to write to the user's buffer.	
IRP$L_MEDIA	First I/O status longword. The I/O postprocessing routine copies the contents of this field, also called IRP$L_IOST1, into the IOSB.	

Table 6-3 (Cont.) **Contents of the I/O Request Packet**

Field Name	Description
IRP$L_IOST2	Second I/O status longword. The contents of this field are also copied into the IOSB during I/O postprocessing.
IRP$B_CARCON	Carriage control.
IRP$L_ABCNT	Accumulated bytes transferred. This field is read and written by IOC$IOPOST after a partial virtual transfer.
IRP$L_OBCNT	Original transfer byte count. This field is read by IOC$IOPOST to determine whether a virtual transfer is complete, or whether another I/O request is necessary to transfer the remaining bytes.
IRP$L_SEGVBN	Virtual block number of the current I/O segment. This field is written by IOC$IOPOST after a partial virtual transfer.
IRP$L_DIAGBUF	Diagnostic buffer address in system address space. EXE$QIO copies the buffer address into this field if the following three conditions exist: • The I/O request call specifies this address. • A diagnostic buffer length is specified in the driver dispatch table. • The process has diagnostic privilege.
IRP$L_SEQNUM	I/O transaction sequence number. If an error is logged for the request, this field contains the universal error log sequence number.
IRP$L_EXTEND	Address of the I/O request packet extension (IRPE). FDT routines write an extension address to this field when a device requires more context than the IRP can accommodate. This field is read by IOC$IOPOST. The IRP$V_EXTEND bit in the IRP$W_STS field is set if this extension address is used.
IRP$L_ARB	Access rights block (ARB) address. This block is located in the PCB and contains the process privilege mask and UIC.
IRP$L_KEYDESC	Address of encryption descriptor.

All IRPs for a particular UCB are linked together through the UCB$L_IOQFL and UCB$L_IOQBL fields. All IRPs to be postprocessed are linked together by the global cell IOC$GL_PSFL.

An IRP may also be used as an AST control block. For more information, see Section 6.3.5.

6.2.2 The Function Decision Table

Every device driver contains a **function decision table** (FDT) that lists all the valid function codes for the device, and associates valid codes with the addresses of I/O preprocessing routines called **FDT routines**.

Allocated from nonpaged pool as part of the driver image, an FDT is pointed to by an associated driver dispatch table. The FDT routines execute in process context, and they access process space (P0 and P1). There are five major FDT functions:

- Access (and Create)
- Deaccess
- Modify (and Delete)
- Mount
- Read (and Write)

The FDT routines for file system functions are in the SYS module SYSACPFDT.

When a user process calls the $QIO system service, the system service uses the I/O-function code specified in the request to traverse the FDT. The FDT contains information for the device-dependent portion of I/O preprocessing, and one or more of these routines is selected for execution.

FDT routines complete the I/O preprocessing phase by performing setup and initialization functions. For example, for virtual Read and Write requests, the FDT routines initialize two fields in the IRP. The IRP$L_OBCNT field contains the total number of bytes in the original request, and the IRP$L_ABCNT field, initialized to 0, accumulates the total number of bytes actually transferred. The function routines then queue the IRP to the XQP for processing.

FDT routines also detect total mapping failure (that is, the information in memory that describes the sections of the disk to be accessed is not sufficient).

FDT routines are accessed and run in the context of the process that requested the I/O. They execute at IPL$_ASTDEL, which prevents ASTs from being delivered to the process but allows the FDT routine code to be pageable. ASTs must be blocked to prevent process deletion because the address of the allocated I/O packet is held in a register and is not recorded elsewhere in the system.

QIO processing is also performed in process context. While QIO processes the request or while the FDT routines are executing at IPL 2, the process can be preempted; therefore, context can be lost.

Except for two special cases, FDT entries consist of three longwords: two longwords containing a 64-bit function mask and one longword containing the address of an action routine.

The I/O function code requested by the user is 16 bits long and is encoded in two fields:

- The first 6 bits (bits 0-5) contain the function code.
- The remaining 10 bits (bits 6 thru 15) contain the modifier code.

The 6-bit function code is used as a bit number into the 64-bit masks. If the bit number corresponding to the I/O function is set in the mask, QIO dispatches to the action routine.

There are two special cases; each consists of a 64-bit mask. The first of these contains bits set to identify legal I/O functions for this device, which allows QIO to validate the function code.

The second identifies buffered I/O functions. This mask prevents the duplication of code that would otherwise exist in many routines.

Every function decision table has this format: two special masks followed by a variable number of 3-longword entries. No special entry denotes the end of the FDT.

Figure 6-5 shows the format of a function decision table.

6.2.3 The Driver Dispatch Table

The driver dispatch table (DDT) contains the address of the function decision table as well as other driver-specific information such as the addresses of the entry points of standard routines within the driver. It is pointed to by the UCB for the device.

6.3 Internal Dispatching

Internal dispatching is a part of the I/O preprocessing phase. Dispatching begins with a call to the $QIO system service.

Issuing a QIO results in a call to the SYS$QIO system service vector. The vector contains an entry mask, a CHMK #QIO instruction, and a RET instruction. Execution of the CHMK instruction causes an exception, which is vectored through the system control block to the change mode dispatcher.

The exception mechanism changes the access mode to kernel mode and places the CHMK operand, the #QIO, on top of the stack. All the registers are saved by the call, with the exception of R0 and R1.

The change mode dispatcher obtains the exception code and verifies that it is legitimate. It checks that the argument list is the right length for the QIO and that the argument list is may be read in the access mode from which the system service request was issued. The change mode dispatcher then calls the QIO service routine EXE$QIO.

Figure 6-5 Layout of a Function Decision Table

Art number: ZK921-82 in Driver book

6.3.1 $QIO System Service Dispatching

QIO preprocessing begins in the SYS module SYSQIOREQ. The EXE$QIO routine in the SYSQIOREQ module performs the device-independent preprocessing of an I/O request and calls a driver's FDT routines to perform device-dependent processing. Once the operation has been started, control returns to the caller, who can synchronize I/O completion in one of three ways:

1. Specifying the address of an AST routine to be executed when the I/O completes.
2. Waiting for the specified event flag to be set.
3. Checking the specified I/O status block (IOSB) for a completion status. The IOSB is an 8-byte block into which a device-dependent system status code can be written.

There are twelve parameters to the QIO system service: six device-independent parameters and six device-dependent parameters defined by the actual device. EXE$QIO processes only the device-independent parameters; it defines an FDT routine to process the device-dependent parameters.

To validate the I/O request, the following function-independent parameters are verified:

- The event flag number (EFN) must be legal. EFN 0 is the default. Local event flags process more quickly than common event flags because the local event flags are actually contained in the PCB. Only the addresses of the common event flags are contained in the PCB, and therefore, an extra level of indirection is incurred.
- The access mode must be legal. This mode applies to the channel over which I/O has been requested.
- A UCB must be assigned. The UCB must match the device that has been requested.
- The UCB status word is checked to ensure that the online bit is set.
- The I/O function code, which is validated by the FDT, must be legal for the device.
- The IOSB must be writable in the mode in which the QIO was issued. If the I/O request specifies an IOSB to receive final I/O status infromation, EXE$QIO determines whether the process issuing the request has write access to the status-block locations specified. If the process has write access, EXE$QIO fills the IOSB with zeros. If the process does not have write access, the procedure terminates the request with an error status.
- The DIOCNT or BIOCNT quota is checked and updated. IPL is raised to IPR$_ASTDEL to prevent process deletion.

  EXE$QIO determines whether satisfying the I/O request will cause the process to exceed its quota of outstanding direct or buffered I/O requests. If the requests remain under quota, the system service allows I/O preprocessing to continue. If either quota is exceeded, EXE$QIO checks the resource wait flag (the PCB$V_SSRWAIT bit in the PCB$L_STS field).

  If the flag is clear, EXE$QIO aborts the I/O request. If the flag is set, the process is placed in a wait state until the number of requests drops below quota. When this occurs, process execution resumes, at which time EXE$QIO charges process quotas as appropriate for the requested operation.

After the request is validated, it is synchronized with any pending access or deaccess operations on the channel. The channel cannot be closed until outstanding QIOs are completed.

The I/O request packet is then allocated from nonpaged pool. Before EXE$QIO actually allocates the IRP, it raises the IPL of the processor to IPL$_ASTDEL to block any other asynchronous activity in the process. The new IPL prevents possible termination of the process; process termination cause the operating system to lose track of the system memory allocated to the IRP.

To save time, EXE$QIO first tries to allocate an IRP from a **lookaside list** containing preallocated IRPs. The EXE$ALLOCIRP routine in the MEMORYALC module handles this function. If no preallocated packets exist, the procedure calls a routine to allocate an IRP from general nonpaged pool. This allocation routine synchronizes with the rest of the system at IPL$L_SYNCH so it can allocate the memory needed.

To keep track of I/O requests outstanding on the channel, the channel I/O count field (CCB$W_IOC) is incremented in the CCB.

The IRP is then filled in; the function independent parameters and process information are copied to the I/O packet. The I/O function code is validated against process privilege and device characteristics.

If a user AST has been specified for notification of AST completion, the AST count quota field (PCB$W_ASTCNT) is checked and decremented, and the AST quota update flag (ACB$V_QUOTA in the IRP$B_RMOD field) is set.

After the IRP is filled in, the driver's FDT routines that correspond to the specified function are called. At this point, all device-independent processing is done, and device-dependent processing begins.

Figure 6-6 shows the user, VMS executive, and XQP images that are executed while the XQP processes an I/O request.

Figure 6-6 Images Used to Process an I/O Request

```
User image      VMS executive image            XQP image
----------      -------------------            ---------

  $QIO ------------> EXE$QIO::
                            |
                            |
                            V
                FDT routines detect
                an ACP function or
                total  map failure,
                so the XQP is needed
                            |
                            |
                            V
           +--> EXE$QXQPPKT:: sends   _____>      DISPATCH::
           |     an AST to the XQP  ___\\     1)  Does the requested
           |                                      function
           |                                  2)  Sets up the IOPOST
          y|                                      interrupt or calls
          e|                                      IOPOST directly
          s|                 <------------- 3)  Returns
           |
           |        IOPOST::
           |     Is the QIO incomplete
           <--- or is bad block
                handling needed?
                            |
                   (no)     |
                            V
                1)  Post results to
                    the user
                2)  Cleanup
        <------- 3)  Performs an REI
```

6.3.2 Function Decision Table Dispatching

When the $QIO system service has executed all the device-independent code, QIO searches the device database and finds the correct function decision table address. The channel control block, located through the channel number argument, contains the UCB address. The UCB contains the address of the driver dispatch table (DDT), which contains the address of the function decision table.

QIO scans the function decision table, starting at the third entry (that is, the first entry after the two special cases), using the 6-bit function code as a bit number into each mask. If the bit is set, QIO calls the routine that the bit represents. The routine must then finish filling in the device-dependent part of the I/O packet.

The most frequently used functions are at the front of the table, so scanning is fast and very efficient.

If the FDT routine returns to QIO, QIO advances to the next entry in the FDT and checks the I/O function code and mask. If the bit is not set, QIO again advances to the next FDT entry. If the bit is set, however, QIO dispatches to the indicated routine.

FDT routines send information (such as whether the I/O is direct or buffered) to the driver in the driver-dependent part of the IRP. At this point, the FDT routines determine whether an I/O transfer from the disk is needed. If a transfer request has been specified (that is, a virtual Read or Write operation), then the WCB is read to obtain the mapping information to see if the request can be processed as is or whether XQP intervention is necessary.

When a transfer request if processed, one of two cases exists:

- The map information in memory is sufficient to map the request either successfully or partially.
- The map information is totally insufficient (total map failure).

In the first case, the available information is queued to the driver, and the results are posted to the user. If a partial mapping occurred, more mapping information is sought.

In the second case, the XQP must obtain new mapping information by turning the current window. When the new mapping information is obtained, it is queued to the driver's routine to start I/O (EXE$QIODRVPKT).

Figure 6-7 shows the logic that determines the action the file system takes when a transfer request is initiated.

However, if a non-transfer request was specified (that is, a Deaccess operation (if the process was not the last writer) or an access to an already accessed file), an XQP packet is built instead.

Figure 6-7 XQP Logic for an I/O Transfer Request

```
XQP turns  <-------  Can the file be   <---+
window       no    successfully mapped?    |
   |                        |              |
   |                        | yes          |
   |                        |              |
   |                        V              |
   +---------------->  Queue to driver     |
                            |              |
                            |              |
                            V              |
                         IOPOST            |
                            |              |
                            |              |
                            V              |
                     More to map?   -------+
                            |       yes
                            | no
                            V
                     Post to user
                      when done
```

6.3.3 Building the XQP I/O Packet

The IRP sent to the XQP contains the address of an **XQP I/O buffer packet** (AIB) in the IRP$L_SVAPTE field. This XQP packet is built in the SYSACPFDT routine BUILDACPBUF.

The AIB is 12 bytes long. The first longword, the AIB$L_DESCRIPT field, points to a vector of buffer descriptors. "Buffer descriptor" refers to a user area into or from which information is transferred. The actual user buffers are copied into the AIB buffers during FDT processing (see Figure 6-12). They are copied back to the user's area during I/O postprocessing in the BUFPOST routine in the SYS module IOCIOPOST. For more information on I/O postprocessing, see Section 6.6.

The format of an XQP I/O buffer packet is shown in Figure 6-8 and described in Table 6-4. The AIB contains all the data transmitted from the user to the XQP and back during an ACP function.

Figure 6-8 Format of an XQP I/O Buffer Packet

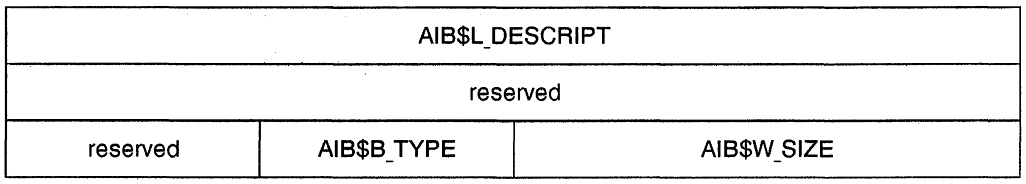

Table 6-4 Contents of an XQP I/O Buffer Packet

Field Name	Description
AIB$L_DESCRIPT	Address of start of descriptors.
AIB$W_SIZE	Size of packet in bytes.
AIB$B_TYPE	Packet type code. This field contains the constant DYN$C_BUFIO for a buffered I/O function.

Before the buffer can be allocated, BUILDACPBUF must ensure that the buffer byte count quota has not been exceeded. The user parameters to the QIO (such as the FIB) are also checked to make sure they can be accessed. Then the buffer is allocated, and the buffer descriptors (ABDs) for each user parameter are inserted in the XQP packet.

Each **complex buffer descriptor** contains an offset to the text data, the size, and the user virtual address of the data. The offset, plus 1, added to the address of the buffer descriptor gives the address of the buffer (the preceding byte is the access mode taken from IRP$B_RMOD). Each possible user buffer has a reserved index in the vector. The indexes are zero origin. The last element reserved corresponds to the read/write attribute user function. All buffers from then on correspond to Read/Write Attribute buffers. IRP$L_BCNT contains the number of buffer descriptors present. (Note that for a window turn, IRP$V_COMPLX is clear, so the above description does not apply).

The fields of the IRP contain the standard infroamtion, and the SVATPE field points to the complex buffer packet. The byte count word in this case indicates the number of buffers; that is, it indicates the number of descriptors and the number of buffers that are in the complex buffer packet.

The complex buffer descriptor consists of the packet header and a list of descriptors. Each descriptor contains the actual size of the buffer. There is an offset pointer to the data text, which is located farther down in the packet.

Figure 6-9 shows how the complex buffer descriptor is constructed.

Figure 6-9 Locating the Complex Buffer Descriptor

```
    I R P                    Complex Buffer Descriptor

                     +------------------------------------+
+-------------+      |       +------------------------+   |
/             /      |       |                        | <----+
/             /      |       /         A I B          /
+-------------+      |       /                        /
| IRP$L_SVAPTE | -----+      +------------------------+ -----+
+------+------+              |       ABD$C_WINDOW     |      |
| BOFF | BCNT |              +------------------------+      |
+------+------+              |       ABD$C_FIB        |      +--A B D
/             /              +------------------------+      |
/             /              |       ABD$C_NAME       |      |
+-------------+              +------------------------+      |
                             |       ABD$C_RESL       |      |
                             +------------------------+      |
                             |       ABD$C_RES        |      |
                           / +------------------------+      |
                          /  |  descriptor for first  |      |
                         /   |       attribute        |      |
                        /    +------------------------+      |
          30 maximum-----    /           .            /      |
                        \    /           .            /      |
                         \   +------------------------+      |
                          \  |  descriptor for last   |      |
                           \ |       attribute        |      |
                             +------------------------+      |
                             |                        |      |
                             |  text area used to     |      |
                             |  contain information   |      |
                             |  to which the above    |      |
                             |  descriptors point     |      |
                             +------------------------+ -----+
```

The ABD may contain a maximum of 35 descriptors. The first five descriptors have special names and uses:

- The first descriptor, ABD$C_WINDOW, is for returning the window pointer. The user does not supply this buffer. Many file system routines use this field differently.

  During FDT processing, the BUILDACPBUF routine sets the window pointer return address to the value in the CCB$L_WIND field.

  When retrieving a request from the XQP queue, the GET_REQUEST routine zeros the window pointer return length (except during window turns) so that the value is not returned.

When accessing a file, the MAKE_ACCESS routine restores the window pointer return length to 4 and returns the window pointer.

If an access attempt fails, the ZCHANNEL cleanup routine returns a zero for the window pointer.

- The second descriptor, ABD$C_FIB, contains the user's FIB. The FIB travels in two directions: both from the user to the XQP, and from the XQP back to the user. It is copied into the LOCAL_FIB portion of the XQP impure area by the GET_FIB routine. The updated FIB is copied back to the FIB buffer by the IO_DONE routine.
- The third descriptor, ABD$C_NAME, contains the filename buffer. It is passed as the input to the PARSE_NAME routine from the Enter and Find functions to parse the user's filename into the internal name block. The COPY_NAME routine (called from the Create and Find functions for spooled devices) copies the filename buffer into the result string buffer. It also sets the result string length buffer value.

  The filename string travels only in one direction: from the user to the XQP. Its counterpart, the resultant name string, is what is sent back to the user from the XQP. For efficiency, the IO_DONE routine clears the filename return length to prevent it from being written back.

  For quota file operations, the file name buffer is used to pass a quota file transfer block (DQF). For operations on a spooled device, FDT processing places the username and account in the filename string to be placed in the file header.
- The fourth and fifth entries, ABD$C_RESL and ABD$C_RES, are descriptors for the result length and the result string, respectively. The RETURN_DIR routine, called from the Enter and Find functions, returns the name from the DIR_ENTRY and DIR_VERSION routines into the result string buffer. The result string length buffer is also set. The result string is itself passed to the PARSE_NAME routine from the Find function when the XQP processed a wildcard search. Quota file operations call the RET_QENTRY in the QUOTAUTIL module to return the quota record (DQF) into the result string buffer. The result string length is set here.

If a user attribute buffer exists, a Read/Write Attributes function is performed. The Access function performs an attribute read. The Create, Deaccess, and Modify functions perform an attribute write. The IO_DONE routine sets the IRP$L_BCNT field during non-read operations so that the attributes, which are no longer needed, are not written back to the user buffers for optimization reasons.

The attribute list sometimes contains placement data (processed for compatibility) when the FIB$V_ALLOCATR field is set. The GET_LOC_ATTR routine, called the the Create and Modify functions, scans the user's attribute list for placement data and copies it into standard format in the FIB.

The fields of a complex buffer descriptor are shown below in Figure 6-10 and described in Table 6-5.

Figure 6-10 Format of a Complex Buffer Descriptor

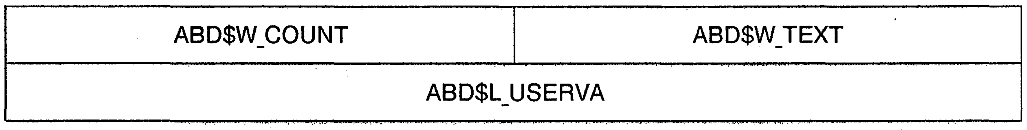

Table 6-5 Contents of a Complex Buffer Descriptor

Field Name	Description
ABD$W_TEXT	Word offset to the data text in the text area. Figure 6-11 shows the format of a single data text entry.
ABD$W_COUNT	Length of text in bytes.
ABD$L_USERVA	User virtual address of text (P0 address). This address is needed to post buffers back to the user.

The format of a data text entry is shown below in Figure 6-11. The buffer descriptors point to this area. The figure shows the prefix byte in front of the data buffer. This byte normally contains the access mode against which the buffer is validated. Generally, that is the mode of the caller, with the exception of the very first buffer, which is used to access the user's channel control block when the CCB needs to be adjusted by the file system. In that case, the prefix byte contains the kernel mode code.

For the attribute buffers, the prefix byte contains the attribute code as the complex buffer packet travels from the user to the XQP. When the complex buffer packet is sent back to the user, the prefix byte has been changed to contain the access mode.

Figure 6-11 Format of a Data Text Entry

```
+-----------------+----------+---------+ Access mode from IRP$B_RMOD
|                            |\\\\\\\\\|  or attribute descriptor
|       t e x t   d a t a    +---------+ code
|                                      |
|                                      |
+-----------------+--------------------+
```

Figure 6-12 shows how user information is copied from user context to the XQP. In this case, a FIB is copied from the user's stack in P1 space into the data text portion of the ABD. A FIB descriptor is created to describe and locate the data.

The text offset field points to the data text portion of the ABD. The access mode area of the data text entry contains a 1, which indicates executive mode.

The module GETFIB performs the copy operation from the data portion of the ABD into the LOCAL_FIB portion of the XQP impure area. The CURRENT_FIB field, which points to the FIB currently in use, points to the LOCAL_FIB field, which points to the primary FIB of this operation. The count field of the ABD contains the number of bytes in the FIB text, which are copied to LOCAL_FIB.

The number of descriptors is placed in the IRP$L_BCNT field, and the number of bytes charged to the buffer byte count quta is written into the IRP$L_BOFF field. In addition, the COMPLX, FILACP, and VIRTUAL bits are set in the IRP$W_STS field. Finally, the original UCB address in CCB$L_UCB is placed into IRP$L_MEDIA, and IPL is set to IPL$_SYNCH.

6.3.4 Checking the Volume Status

The FDT routines ensure that the volume has the correct state for the request. For a request to succeed, the volume must not be in the following states:

- Marked for dismount
- Not mounted
- Mounted with the /FOREIGN qualifier (that is, the volume is not a Files-11 volume)

The check dismount routine CHKDISMOUNT ensures that the volume is not being dismounted. If the DEV$V_DMT bit is set in the UDB$L_DEBCHAR field, the volume has been marked for dismount.

The CHKMOUNT routine checks to ensure that the following states exist:

- The device is mounted. If so, the DEV$V_MNT bit is set in the UCB$L_DEVCHAR field.
- The device is not a member of a shadow set.
- The device is not in the dismount state. If so, the UCB$V_DISMOUNT bit is set in the UCB$W_STS field.
- The volume is not mounted foreign.

Once the volume checks succeed, the volume transaction count (contained in the VCB$W_TRANS field) is incremented. This update is normally done for the volume describing the desired UCB, but it may be done to the UCB on which a file is open if the WCB so indicates. The IRP$L_UCB field is updated to this value. The IRP$L_MEDIA field is updated to this UCB if the device is not spooled.

Figure 6-12 Passing User Information to the XQP

```
User's stack in
   P1 space
+-------------+
|             |
/             /             XQP Buffer Descriptors
/             /          +---------------------------+
+-------------+          /                           /
| User's copy |  --+      /                           /
| of FIB      |    |      +-------------+-------------+  ----+
+-------------+    |      |   n bytes   | Text offset | -+   +-- FIB
/             /    |      +-------------+-------------+  |   |descriptor
/             /    |      |User virtual address of FIB|  |   |
+-------------+    |      +---------------------------+  |  -+
                   |      /                           /  |
                   |      /                           /  |
                   +----> +--------------------+------+ <+
                         /|     FIB from       |  1   | access mode of
                     n__/ |   user's context   +------+ EXEC
                     |  \ |                           |
                     |   \+---------------------------+
XQP impure area      |
+-------------+ <--  +--- CTL$GL_F11BXQP
|             |      |
+-------------+      |
| CURRENT_FIB |  -+  |
|             |   |  |
+-------------+   |  |
|             |   |  |
+-------------+<--+  |
|  LOCAL_FIB  |\-----+
|             |/
+-------------+
|             |
+-------------+
|  SECOND_FIB | for secondary file operations
|             |
+-------------+
|  LOCAL_ARB  | local copy of caller's access rights block
+-------------+
/             /
/             /
+-------------+
```

6.3.5 Queuing the I/O Packet to the XQP

It is crucial for proper synchronization that the XQP dispatcher be called via AST scheduling. XQP packets may be queued, or dispatched, to the XQP by the following two routines:

- IOC$WAKACP in the SYS module IOCIOPOST via a special kernel AST. The special kernel AST ensures that the IRP won't be invalidated by process deletion between the time IOPOST is exited and the time a normal KAST could be delivered to the process.
- EXE$QIOACPPKT in the SYS module SYSQIOREQ via a normal kernel AST.

After the FDT routines have completed filling in the device-dependent parameters, the last entry usually contains a branch instruction to the EXE$QIOACPPKT routine in the SYSQIOREQ module to perform one of the following actions:

- Terminate the current request with an error status.
- Put the request in the driver queue, and return an appropriate status to the user.
- Signal that the I/O request has been completed, and return an appropriate status to the user.

EXE$QIOACPPKT is called at IPL$_ASTDEL to prevent the user's process from being deleted; the IRP cannot be lost before it is inserted in the XQP request queue. The routine generates, within the context of the user's process, a kernel mode AST specifying as the AST routine the value found in the F11B$L_DISPATCH field, which contains the XQP dispatcher address. EXE$QXQPPKT then queues the kernel mode AST to the XQP dispatcher.

To avoid allocating an AST control block (ACB), the CDRP extension to the IRP is used as an ACB. This area is normally used by the disk class driver when processing disk I/O requests.

The fields of the ACB are illustrated in Figure 6-13 and are described in Table 6-6. Note that the fields of the figure run right to left.

Figure 6-13 Format of the AST Control Block

ACB$L_ASTQFL		
ACB$L_ASTQBL		
ACB$B_RMOD	ACB$B_TYPE	ACB$W_SIZE
ACB$L_PID		

(Continued on next page)

Figure 6-13 (Cont.) Format of the AST Control Block

ACB$L_AST
ACB$L_ASTPRM
ACB$L_KAST

Table 6-6 Contents of the AST Control Block

Field Name	Description	
ACB$L_ASTQFL	AST queue forward link. This field links the ACB into the AST queue for the process; the listhead for the queue is the PCB$L_ASTQFL field.	
ACB$L_ASTQBL	AST queue backward link. This field links the ACB into the AST queue for the process; the listhead for the queue is the PCB$L_ASTQBL field.	
ACB$W_SIZE	Structure size in bytes.	
ACB$B_TYPE	Structure type code. This field should contain the constant DYN$C_ACB.	
ACB$B_RMOD	Access mode of the requestor. The following fields are defined within ACB$B_RMOD:	
	ACB$V_MODE	Mode for final delivery. This field contains the access mode (0-4) in which the AST routine is to execute. This field occupies bits 24 and 25.
	ACB$V_PKAST	Piggyback special kernel AST. This is bit 28.
	ACB$V_NODELETE	ACB is not deallocated after the AST is delivered. This bit generally indicates that the ACB is part of a larger structure. This is bit 29.
	ACB$V_QUOTA	Process AST quota (PCB$W_ASTCNT) has been updated. This is bit 30.
	ACB$V_KAST	Special kernel AST. This is bit 31.
ACB$L_PID	Process ID of the process to receive the request, from IRP$L_PID.	
ACB$L_AST	AST routine address. EXE$QXQPPKT writes into this field the address of the DISPATCH routine from (@CTL$GL_F11BXQP) + F11B$L_DISPATCH.	

Table 6-6 (Cont.) Contents of the AST Control Block

Field Name	Description
ACB$L_ASTPRM	AST parameter. This field contains the address of the IRP.
ACB$L_KAST	Internal kernel mode transfer address. This field contains the address of the EXE$QXQPPKT routine.

The following code fragment shows a portion of EXE$QXQPPKT in SYSQIOREQ. The XQP packet is queued to the XQP with a normal kernel AST, and the CDRP extension to the IRP is used as an ACB.

```
EXE$QXQPPKT::
    MOVL   G^CTL$GL_F11BXQP, R0                ;XQP queue head address
    MOVAB  CDRP$L_IOQFL(R5), ACB$L_ASTPRM(R5)  ;IRP address is AST parameter
    MOVB   #PSL$C_KERNEL!ACB$M_NODELETE, -     ;Kernel mode---don't delete
           ACB$B_RMOD(R5)                      ;IRP
    MOVL   PCB$L_PID(R4), ACB$L_PID (R5)       ;Copy PID
    MOVL   F11B$L_DISPATCH(R0), ACB$L_AST(R5)  ;XQP dispatcher address
    MOVL   #PRI$_RESAVL, R2                    ;Like waiting for a lock
    BSBW   SCH$QAST                            ;Queue the AST
    RSB                                        ;And return
```

This code fragment shows a portion of IOC$WAKACP in IOCIOPOST. Like the code in EXE$QXQPPKT, the CDRP extension to the IRP is used as an ACB; however, the XQP packet is queued to the XQP with a special kernel-mode AST instead of with a normal kernel AST.

```
IOC$WAKACP::
       .
    .
    .
    TSTL   AQB$L_ACPPID(R2)                    ;No PID if XQP
    BEQL   XQP                                 ;Equal, then branch to XQP
       .
    .
    .
XQP::
    PUSHL   R5                                 ;Preserve R5
    MOVAB   IRP$L_FQFL(R3), R5                 ;Get temp ACB address in R5
    MOVB    #ACB$M_KAST, ACB$B_RMOD(R5)        ;Note as special kernel AST
    MOVL    IRP$L_PID(R3), ACB$L_PID(R5)       ;Copy PID of process
    MOVAB   W^EXE$QXQPPKT, ACB$L_KAST(R5)      ;Address of queuing routine
    CLRL    R2                                 ;No priority increment
    BSBW    SCH$QAST                           ;Queue the AST
    POPL    R5                                 ;Restore R5
    RSB                                        ;And return
```

6.4 XQP Code Execution

When the kernel AST queued to the XQP dispatcher begins to execute, the code in the F11BXQP image is executed. This code is entered from three routines:

- EXE$QIO via EXE$QXQPPKT—This routine calls the XQP to perform ACP I/O functions and window turns for IO$_READVBLK/WRITEVBLK with total map failure.
- IOC$IOPOST via IOC$WAKACP—This routine calls the XQP to perform dynamic bad block handling and window turns for the next segment of discontiguous long virtual I/O with total map failure.
- DIRPOST via IOC$WAKACP—This routine calls the XQP to queue IO$_DEACCESS on an idle channel.

The DISPATCH routine is the XQP dispatcher routine. The argument to this routine (that is, the AST parameter) is the IRP.

Figure 6-14 illustrates the flow of code through the XQP after the AST has been sent to DISPATCH.

The routine also sets up a register (R10, called the base register) to point to the XQP impure area, the address of which is contained in the cell CTL$GL_F11BXQP). All XQP routines assume that R10 points to the XQP_QUEUE offset in the XQP storage area. The IRP is then queued onto a per-process queue in the impure area called XQP_QUEUE. F11B$Q_XQPQUEUE points to this queue.

The PCB$B_DPC cell is incremented to prevent process deletion while any file system request is being processed; I/O cannot be returned to a nonexistent process. This field must contain a 0 before the EXE$DELPRC routine, in the SYS module SYS$DELPRC, can proceed with process deletion. EXE$DELPRC waits at IPL 0 to allow kernel ASTs to be delivered so that pending file system requests can complete. Similar code in the process suspension service prevents a process from being suspended until pending file system requests are completed.

Process suspension must be prevented while file system requests are active; otherwise, random synchronization locks could be held indefinitely, which could potentially hang an entire VAXcluster. On the other hand, process deletion must be blocked while a file system request is being processed to prevent problems that could be caused by partially -completed operations.

If there are no other requests being processed, which is the normal case, the routine enables the special XQP channel by writing 1 into the CCB$B_AMOD field so it appears to be a normal kernel mode channel; the CCB$B_AMOD field contains the current access mode (kernel, or 0) plus 1. The channel thus becomes inaccessible to any other process at any mode because the privilege check for channels in IOC$VERIFYCHAN performs a signed comparison against access mode. The system rundown routine, EXE$RUNDWN, in the SYS module

Figure 6-14 XQP Code Flow

```
           EXE$QIO issues an AST to the XQP
              in the routine DISPATCH
                       |
                       V
               Put IRP on XQP_QUEUE
                       |
                       V        (yes)
                   Is XQP busy? ----->  then return
                       |
                       V (no)
            Switch to XQP internal stack
                       |
                       V          (queue empty)
+-------------> Get next request -----> restore kernel -----> return
|                 from XQP_QUEUE              stack
|                      |
|                      V
|     no    Do requested function     yes
|     +---  Physical READ or WRITE?  ---+
|     |                                 |
|     V                                 V
|  Post results with           Post results by
|   an IOPOST AST            calling IOPOST via JSB
|     |                                 |
|     V                                 V
+-----+---------------------------------+
```

SYSRUNDWN, also does signed comparisons against access mode to determine if a given channel should be deassigned. When the XQP is not actively processing a request, the special XQP channel contains a negative access mode (that is, -1), which prevents it from being deassigned.

6.4.1 Dispatching a Request

The main dispatch routine, DISPATCHER, is called from DISPATCH. This routine dequeues a request, executes it, and signals the user when the request has been completed. The XQP uses its private kernel stack to process the requests. After completing the first request, it attempts to dequeue another request and process it.

The actual requests to be processed are obtained by GET_REQUEST. The routine first initializes the impure area, which involves zeroing the impure area and setting the user request status USER_STATUS to 1 (or success). The per-process buffer (BFR_LIST) queue heads are set to empty lists. Also, PMS monitoring is started.

The pointers to the current UCB, FIB, and WCB are obtained from the current I/O packet and are written to the CURRENT_FIB, CURRENT_UCB, and CURRENT_WINDOW cells of the XQP impure area. If the low bit of the pointer to the window (the IRP$L_WIND field) is set, a Deaccess function is pending on the file, and so CURRENT_WINDOW is zeroed.

The values for CURRENT_FIB, CURRENT_UCB, and PRIMARY_FCB are set if a window exists; a window does not exist for Access, Create, or Mount functions. Values for CURRENT_VCB, CURRENT_RVT, and CURRENT_RVN are also established.

If the I/O request is a normal FCP request and not a window turn (that is, the IRP$V_COMPLX bit is set), the byte count for the window block descriptor (the ABD$C_WINDOW) field is cleared to prevent the I/O completion routines from writing it back.

The SYSPRV flag in the local copy of the access rights block is set if appropriate. The VOLOWNER and GROUPOWNER cleanup flags are set, as well as the SYSPRV cleanup flag if SYSPRV, BYPASS, or READALL privileges are set.

Returning to the main flow of DISPATCHER, the file system function code is obtained from the IRP$V_FCODE field. The minimum number of buffers needed for the function is obtained in the GET_REQD_BUFR_CREDITS routine (see Chapter 5.

The Read and Write Physical Block, ACP control, and Mount functions are performed directly. All other functions must first ensure that the activity block lock (in the BLOCK_LOCKID cell), which blocks all XQP activity on the volume, is free by calling the routine START_REQUEST in the module DISPATCH.

START_REQUEST sets IPL to SYNCH and tests the VCB$L_BLOCKID field to see if a blocking lock already exists. If no blocking lock is currently held on the volume, the activity count in RVT$W_ACTIVITY is incremented by 2 (so that the count remains even), and IPL is lowered to 0.

If a blocking lock already exists, IPL is immediately lowered to 0, and the routine BLOCK_WAIT in the module LOCKERS is called, which waits for the volume blocking lock to be released. When the blocking lock becomes available, START_REQUEST is called again.

DISPATCHER then calls the appropriate routines to process the designated file system function. After the function has compled, PERFORM_AUDIT is called. See Section 6.5.3 for more information on PERFORM_AUDIT.

In addition, cleanup is performed. If the status indicates success, then a normal cleanup is done; any error invokes ERR_CLEANUP.

DISPATCHER then calls the routines that handle termination of I/O processing. For more information, refer to Section 6.6.

6.4.2 Processing in Secondary Context

Some file system functions require what are called **secondary functions.** A secondary function is a normal file system function that is generated by, or on behalf of, another file system function, called a **primary function.** The primary function is not necessarily dependent on the results of the secondary function in order to complete.

To simplify matters when a secondary function is necessary, the context of the primary function, contained in the cells of the impure area delimited by CONTEXT_START and CONTEXT_END, is copied to the area delimited by CONTEXT_SAVE and CONTEXT_SAVE_END. The secondary function can then process as if it were a primary function. Saving the primary context eliminates having to allocate and queue another IRP, which makes processing more efficient. The secondary save area allows only one secondary operation nested within the primary.

The routines that perform the context change are SAVE_CONTEXT and RESTORE_CONTEXT in the module GETREQ.

Figure 6-15 shows how the primary context is copied from the primary function area to the context save area.

Figure 6-15 Saving and Restoring Primary Context in the XQP Impure Area

```
     STORAGE_START --> +--------------------------+
                       |            :             |
     CONTEXT_START --> +--------------------------+\
                       |                          | \
+-------restore------> |       context of         |  \----+
|                      |    primary function      |  /    |
|                      |                          | /     |
|    CONTEXT_END  ---> +--------------------------+/      | save
|    CONTEXT_SAVE ---> |                          |       |
|   /                  |         context          | <-----+
+--/                   |        save area         |
  \                    |                          |
   \CONTEXT_SAVE_END   +--------------------------+
                       |            :             |
     STORAGE_END --->  +--------------------------+
```

After all processing for the secondary function has been completed, the primary context is restored. The ERR_CLEANUP routine detects if any processing has been done in secondary context, and cleans up secondary context before switching to primary context.

Secondary context may leave unwritten buffers. However, any serialization locks obtained in secondary context must be released, and any buffers protected by these locks must be written to disk (refer to Chapters 7 and 5). Also, any unrecorded blocks must be recorded before leaving secondary context (refer to Section 6.4.2).

The following functions require the use of the secondary context area:

- Mapping VBNs (that is, forcing a window turn) when the existing window control blocks do not map the desired VBN during a virtual read or write function. MAP_VBN is responsible for mapping the correct VBN. See Section 6.4.2.1 for more information.
- Operating upon the pending bad block file, BADLOG.SYS. The routines responsible for this function are the SCAN_BADLOG and DEALLOCATE_BAD routines. See Section 6.4.2.2 for more information.
- Marking for deletion a file being removed or superseded during a file creation. The CREATE routine handles this function.
- Opening a file from which attributes are being propagated. The CREATE routine also handles this function.
- Opening a file to determine placement. The GET_LOC routine is responsible for handling this function.
- Extending the index file. This function is performed by the EXTEND_INDEX routine.
- Extending or compressing a directory. The SHUFFLE_DIR routine handles this function.

6.4.2.1 Window Turning

A file may contain one or more extents, and the file header contains a pointer to each extent. Each pointer consists of a starting LBN and an extent size (in bytes).

Figure 6-16 shows the virtual and physical representations of a file with nine extents. Extents are virtually contiguous, but they may physically reside anywhere on the disk.

Figure 6-16 Virtual and Physical Representations of a File

```
   virtual              physical structure on disk
representation

 +----------+          +----------+     +----------+
 | extent 1 |           | extent 3 |     |          |
 +----------+          |          |     | extent 2 |
 |          |           +----------+    |          |
 | extent 2 |                            +----------+
 |          |                            | extent 4 |
 +----------+          +----------+     +----------+
 | extent 3 |           | extent 6 |
 |          |           +----------+
 +----------+
 | extent 4 |           +----------+     +----------+
 +----------+          | extent 8 |      |          |
 |          |           +----------+     | extent 7 |
 | extent 5 |                             |          |
 |          |                             +----------+
 |          |                             |          |
 +----------+          +----------+     | extent 9 |
 | extent 6 |           |          |      |          |
 +----------+          | extent 5 |      |          |
 |          |           |          |      |          |
 | extent 7 |           |          |      +----------+
 |          |           +----------+
 +----------+
 | extent 8 |
 +----------+
 |          |
 | extent 9 |
 |          |
 |          |           +----------+
 |          |           | extent 1 |
 +----------+          +----------+
```

For retrieval purposes, these extent pointers reside in a structure in memory called a **window**. The window control block resides in the top portion of the window. Each WCB contains a starting VBN and a variable number of retrieval pointers. The number of pointers may be set with the following methods:

- The DCL command INITIALIZE/WINDOWS=n
- The FAB$B_RTV field at file open time
- The FDL attribute FILE WINDOW_SIZE

- The system paramter ACP_WINDOW[1]

A special type of window that maps the entire file called a **cathedral window**. This type is window is also known as a "segmented window" because multiple WCBs are usually required to contain its mapping information. Each WCB in the chain is called a "window segment."

When a data transfer (a virtual Read or Write operation) is requested, a starting VBN and the size of the request in bytes is given. The file system then maps the VBN to an LBN, which is used to locate the file on disk.

When an extent whose pointer is not in the current window is accessed, the XQP has to read the file header to construct a new window that maps the desired extents. This I/O operation is called a **window turn**. When the file system turns a window, it reads the FCB to find the file header that contains the desired retrieval pointer.

Figure 6-17 shows the mapping information in both the file header and the window control block. The WCB forms the top portion of the window, and it contains mapping information for the first two extents. In this figure, however, if the information contained in extents 6-9 is needed, the XQP must turn the window.

Virtual Read or Write operations are processed by the FDT routines, which force a window turn if the existing WCBs do not map the desired VBN. A request to turn a window is converted into an IO$_READPBLK or IO$_WRITEPBLK operation. The DISPATCHER routine forwards these function codes directly to the READ_WRITEVB routine in the F11X module RWVB.

READ_WRITEVB obtains the necessary information (such as the address of the current window, the block count, and the desired VBN) from the IRP. It obtains the serialization lock on the file and then calls the MAP_VBN routine.

The MAP_VBN routine in the F11X module MAPVBN is responsible for mapping the specified virtual blocks to their corresponding logical blocks, using the supplied window. Because the serialization lock is being held, MAP_VBN can rebuild the FCB (and the extension FCB chain) if the FCB has been modified.

If an Extend operation was performed on a cathedral window being accessed by multiple users, the current window does not map the entire file. In other words, the WCB$V_CATHEDRAL bit is set, but the WCB$V_COMPLETE bit is not. The REMAP_FILE routine in the ACPCNTRL module is called to remap the file to update the mapping information.

[1] Applies only to disks mounted with the /SYSTEM qualifier.

Figure 6-17 Mapping a File with a Window Size of 5

```
   file header          window control block and window

/\/\/\/\/\/\/\/\/\/     +xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx+\
|        .        |      |             WCB              | \
|        .        |      |              .               |  \
|        .        |      |              .               |   \
+-----------------+     +-------------------------------+\   -- fixed
| map information |      | map information for extent 1 | \ / portion
| for extent 1    |     +-----------------------------+  /    of WCB
+-----------------+     | map information for extent 2 | / \
| map information |      +xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx+    \
| for extent 2    |      | map information for extent 3 |      -- window
+-----------------+     +------------------------------+    /
| map information |      | map information for extent 4 |    /
| for extent 3    |      +-----------------------------+   /
+-----------------+     | map information for extent 5 |  /
|        .        |      +-----------------------------+/
|        .        |
|        .        |
+-----------------+
| map information |
| for extent 9    |
+-----------------+
|        .        |
|        .        |
|        .        |
/\/\/\/\/\/\/\/\/\/
```

REMAP_FILE ensures that the entire file is mapped. If necessary, it creates multiple WCBs (window segments) and links them together. While building the window segments, the following situations may occur:

1. The window completely maps the file. In this case, the WCB$V_COMPLETE bit is already set, so REMAP_FILE likewise sets the WCB$V_CATHEDRAL bit and returns.
2. The window was previously complete, but the file was extended. In this case, new window pointers must be added to the last window segment.
3. The window never completely mapped the file. In this case, the FCB chain is traversed to build the associated window segments.

If the file has extension headers, the FCB chain must be searched for the blocks that need to be mapped. The correct FCB is identified either when there are no more FCBs or when the starting VBN of the next FCB is greater than the desired VBN. After finding the correct FCB, three cases may occur when the I/O transfer is attempted:

1. A successful mapping occurs because the current window contains the desired mapping information.
2. A partial mapping occurs because the window contains the starting VBN, but it does not map contiguous extents.
3. Total map failure occurs because the window does not contain any of the desired mapping information.

If the mapping information in the current window is either totally or partially sufficient, the MAP_WINDOW routine is called to map the transfer. MAP_WINDOW maps the specified virtual blocks into their corresponding logical blocks. It calls the system routine IOC$MAPVBLK in the SYS module IOSUBRAMS to perform the actual mapping.

IOC$MAPVBLK searches the WCB list associated with the request to find the mapping pointers that locate the desired VBN. It compares the desired VBN to the starting VBN in the WCB$L_STVBN field. If the desired VBN preceeds the starting VBN, the count of mapping pointers is obtained from the WCB$W_NMAP field.

If the VBN is not contained in the window, total map failure occurs. In this case, a new UCB address (the current UCB address may have been modified by other code) is obtained from the WCB$L_ORGUCB field, which points to the the volume containing the file. The routine then returns.

If the VBN is in this segment, however, the window is scanned, and the count field of each retrieval pointer is subtracted from the current block number. When the retrieval pointer containing the starting VBN is found, the next pointer is also scanned to see if it is contiguous with the one just found in case the transfer request spans two pointers. The maximum number of contiguous retrieval pointers a transfer can span is two.

If the total transfer has been mapped contiguously, the number of bytes mapped and the starting LBN is returned. In addition, the stack pointer is cleared, a status of SS$_NORMAL is returned, interrupts are again allowed, and the routine performs an RSB.

If the transfer has not been completely mapped, the routine returns the number of unmapped bytes, the LBN of the first block mapped, status, and then performs an RSB.

In either case, the IRP is queued to the driver's starting I/O routine.

However, if the map fails because the mapping information in the window is not sufficient, the TURN_WINDOW routine is called to turn the window. This routine contains the code to update window control blocks. The routine handles cases where the file was truncated or extended, and where the WCBs describe VBNs prior to or beyond the desired area. It scans the map area of the supplied file header and builds retrieval pointers in the window until one of the following conditions is met:

- The first retrieval pointer in the window maps the desired VBN.
- The entire header has been scanned.

If no window exists, a new window is created. However, if a window[1] already exists, one of several situations may occur:

1. The window must be turned to map a different portion of the file.
2. The header contains pointers which may be added to the existing window after the existing window is truncated from the beginning.
3. The desired VBN is less than the specified starting VBN and the starting VBN is greater than 1.
4. The window already maps a portion of the header and only the new pointers (which may include a partial map pointer if two contiguous extents were collapsed into one map pointer in the header) have to be mapped.

Figure 6-18 illustrates the first situation, where a window must be turned to map, or point to, a totally different portion of the file because neither the starting VBN nor the desired VBN is contained in the current window. In other words, the window must be turned because of complete map failure. The end result is a window that contains totally new VBNs.

A "scanning window" is contructed, containing the desired VBNs. When this scanning window is complete, all the old VBNs (VBNs 22-99) in the original window are discarded, and the new VBNs (VBNs 100-145) are copied to the window.

1 Does not include cathedral windows.

Figure 6-18 Turning a Window Because of Complete Map Failure

```
         +--------------------+  map area in file header
         |                    |
         |                    |
         +--------------------+


         +--------------------+  file on disk
         |                    |
         |                    |
VBN 22   |xxxxxxxxxxxxxxxxxxxx|
         |                    |\
         |                    | \___ current window maps
         |                    | /    VBNs 22-99
VBN 99   |xxxxxxxxxxxxxxxxxxxx|/
VBN 100  |\\\\\\\\\\\\\\\\\\\\|\
         |\\\\\\\\\\\\\\\\\\\\| \___ desired window maps
         |\\\\\\\\\\\\\\\\\\\\| /    VBNs 100-145
VBN 145  |xxxxxxxxxxxxxxxxxxxx|/
         |                    |
         |                    |
         |                    |
         |                    |
         |                    |
         +--------------------+
```

Figure 6-19 illustrates the second situation, where the header contains pointers which may be added to the existing window after the existing window is truncated from the beginning (or the top).

This situation usually occurs when a file is extended without causing a new file header to be created. The difference between this case and the previous one is that the starting VBN of the file header is contained within the current window, which prevents the window from being discarded totally. In this example, the starting VBN is VBN 18, and the desired VBN is VBN 26. The new window must include both VBNs.

Figure 6-19 Turning a Window to Map Additional Pointers

```
        +--------------------+  map area in file header
        |                    |
        |                    |
        +--------------------+

        +--------------------+
        |                    |
        |                    |
VBN 10  |xxxxxxxxxxxxxxxxxxxx|
        |                    | \
        |                    |  \   current window
        |                    |   -- maps VBNS 10-19
        +- - - - - - - - - - + /
        |                    |----- starting VBN (VBN 18)
VBN 19  |xxxxxxxxxxxxxxxxxxxx|
VBN 20  |                    |
        |                    |
        |                    |
        |                    |----- desired VBN (VBN 26)
        |                    |
VBN 27  |                    |
        |                    |
        |                    |
        |                    |
VBN 99  +--------------------+
```

Figure 6-20 shows how the existing window is truncated from the top, or the beginning of the window. The pointer containing the starting VBN (VBN 18) was part of the old window, but it becomes the beginning of the new window. The new window also includes the desired VBN (VBN 26).

Figure 6-20 Truncating an Existing Window

```
        +--------------------+  map area in file header
        |                    |
        |                    |
        +--------------------+

        +--------------------+
        |                    |
        |                    |
VBN 10  +--------------------+
      / |\\\\\\\\\\\\\\\\\\\\| \
     /  |\\\\\\\\\\\\\\\\\\\\|  \___ truncated portion
 old /  |\\\\\\\\\\\\\\\\\\\\|  /    of old window
window  |xxxxxxxxxxxxxxxxxxxx| /
     \  |                    |\---- starting VBN
VBN 19 \|- - - - - - - - - - |  \
VBN 20  |                    |   \
        |                    |    \-- new window (VBNs 18-27)
        |                    |    /
        |                    |----- desired VBN
        |                    |  /
VBN 27  |xxxxxxxxxxxxxxxxxxxx| /
        |                    |
        |                    |
        |                    |
VBN 99  +--------------------+
```

Figure 6-21 illustrates the third situation, where the desired VBN is less than the specified starting VBN and the starting VBN is greater than 1. This situation occurs when a file is extended and a new file header (an extension header) is created.

In this example, the current window is mapped by extents (VBNs 150-199) from the extension header, and extents mapped by the primary header (VBNs 22-99) are desired. In effect, as the primary file header is read, the window is turned "backwards."

Figure 6-21 Turning a Window to Map a Previous Header

```
        +---------------+  map area in
        |               |   file header
        |               |
        +---------------+

        +---------------+  map area in
        |               |   extension header
        |               |
        +---------------+


        +---------------+  file on disk
        |               |
        |               |
VBN 22  +---------------+
        |\\\\\\\\\\\\\\\|\
        |\\\\\\\\\\\\\\\| \___ desired VBNs
        |\\\\\\\\\\\\\\\| /                                ^
VBN 99  +---------------+/                                 |
        |               |  extents mapped by primary header  |
-------------------------------------------------------------+
        |               |  extents mapped by extension header |
        |               |                                     |
VBN 150 +---------------+                                    V
        |\\\\\\\\\\\\\\\|\
        |\\\\\\\\\\\\\\\| \___ current window
        |\\\\\\\\\\\\\\\| /
VBN 199 +---------------+/
        |               |
        |               |
        |               |
        +---------------+
```

Figure 6-22 illustrates the fourth situation, where the window already maps a portion of the header and only the new pointers (which may include a partial map pointer if two contiguous extents were collapsed into one map pointer in the header) have to be mapped.

In this example, VBNs 1-100 are mapped by a single contiguous extent, and VBNs 101-105 are mapped by a second single contiguous extent. If the file is extended contiguously, the new VBNS may also be mapped by the second extent.

Figure 6-22 Turning a Window to Map a Contiguous Extent

```
        +---------------------+  map area in file header
        | LBN            100  |
        | LBN              5  |
        +---------------------+

VBN 1   +xxxxxxxxxxxxxxxxxxxxx+\  file on disk
        |                     | \
        |                     |  \
        |                     |   \
        |                     |    \
        |                     |     \__ first single extent
        |                     |     /
        |                     |    /
        |                     |   /     current window maps
        |                     |  /      VBNs 1-105
        |                     | /
VBN 100 +- - - - - - - - - - -+/
        |                     |\
        |                     | \-- second single extent
        |                     | /
VBN 105 |xxxxxxxxxxxxxxxxxxxxx|/
        |\\\\\\\\\\\\\\\\\\\\\|\
        |\\\\\\\\\\\\\\\\\\\\\| \___ extend the file
        |\\\\\\\\\\\\\\\\\\\\\| /    contiguously
VBN 110 +---------------------+/
```

Figure 6-23 shows how the file system efficiently collapses, or combines, the contiguous extents into a single extent. The second pointer in the file header now reflects the addition of the new VBNs.

Figure 6-23 Collapsing the Contiguous Extents

```
        +---------------------+  map area in file header
        | LBN             100 |
        | LBN              10 |
        +---------------------+


VBN 1   +xxxxxxxxxxxxxxxxxxxxx+\  file on disk
        |                     | \
        |                     |  \
        |                     |   \
        |                     |    \
        |                     |     \__ first single extent
        |                     |     /
        |                     |    /
        |                     |   /     current window maps
        |                     |  /      VBNs 1-110
        |                     | /
VBN 100 +- - - - - - - - - - +/
        |                     |\
        |                     | \
        |                     |  \
        |                     |   \ -- second single extent
        |                     |   /
        |                     |  /
        |                     | /
VBN 110 +xxxxxxxxxxxxxxxxxxxxx+/
```

After the new window has been initialized, the new window pointers are constructed in a buffer. They are copied into the WCB at IPL$_SYNCH to synchronize with other FDT routines trying to map virtual requests.

After TURN_WINDOW returns, MAP_WINDOW again tries to obtain the mapping information.

When control is returned to READ_WRITEVB, the routine checks to see if the IRP$V_VIRTUAL bit is set and if this operation affects a reserved file (the index file or the bitmap file). For a cluster, all cached buffers are invalidated. The appropriate lock (allocation or serial) is obtained so that the sequence number in the value block is updated. On a single node, the buffers are purged from the cache outright.

Once the block has been mapped, the IRP is requeued to the driver for which it was originally intended for I/O. REQUEUE_REQ, in the module REQUEU, translates the LBN into the corresponding physical block number and converts the I/O function code into the appropriate physical function. The number of unmapped blocks is deducted from the byte count.

If the transfer was only partially mapped, the number of unmapped bytes is subtracted from the value in the IRP$L_BCNT field, and the byte count is rounded to the next block boundary.

If the transfer was totally mapped, the UCB$L_MAXBCNT is checked for the largest transfer allowed. If this value is still under the limit in the IRP$L_BCNT field, then the IOC$CVTLOGPHY routine is called to convert to a logical block to a physical block. After this conversion, the EXE$INSIOQ routine is called to queue the IRP back to driver.

If the IRP$V_VIRTUAL bit is not set, an I/O error has occurred in a file sent for bad block processing[1] . The bad block bit in the FCB is set, and the bad block itself is entered in the bad block log.

6.4.2.2 Dynamic Bad Block Processing

A file is found to have bad or defective blocks when an attempt is made to read or write the file, and an error occurs. The file system sets the bad block bit in the file header. Dynamic bad block processing occurs when the file is deleted.

Blocks declared as bad will likewise be handled by the IO$_READPBLK or IO$_WRITEPBLK functions to be sent to the XQP. Again, the DISPATCHER forwards these I/O function codes directly to READ_WRITEVB. Bad block processing is invoked when the FCB is flagged as having bad blocks. If the virtual bit in the IRP$L_IOST1 field is not set, there is a parity, format, or datacheck error in the file.

The MARKBAD_FCB routine in the RWVB module sets the bad block bit (the FCB$V_BADBLK) in the indicated FCB. Setting FCB$V_BADBLK causes the Deaccess function to set the FH2$V_BADBLOCK bit in the file header. Likewise, INIT_FCB2, which initializes the FCB according to the given file header, sets the FCB$V_BADBLK bit if the FH2$V_BADBLOCK bit is set. Setting FH2$V_BADBLOCK, in turn, causes the DELETE_FILE routine to send the file to the bad block scanner for deletion.

Normally, when a file is deleted, the mapped blocks are returned to the storage bitmap. If the badblock flag in the header is set, the routine SEND_BADSCAN (in the SNDBAD module) sends a message through the special mailbox (ACP$BADBLOCK_MBX) created by INIT_FCP during SYSINIT, and it specifies the UCB and FID of the file to be deleted. If the message is sent successfully, a request is made for a process called BADBLOCK_SCAN.

The bad block scanner contains all privileges, and its UIC is [1,3]. Its job is to scan the deleted file to locate the bad blocks. Blocks that do not have errors after this scan are returned to the storage bitmap, and those that do have errors are appended to BADBLK.SYS (by moving the map pointer from the deleted

[1] Reserved files are not subject to dynamic bad block processing.

file to BADBLK.SYS).[1] To find the defective blocks, the bad block scanner runs BADBLOCK.EXE in the BADBLK facility.

The main BADBLOCK processing routine, MAIN_BAD (in the BADBLK module GETREQ) reads each message from the bad block mailbox. For each, it resets the UCB address in a CCB it holds for that purpose to the UCB address of the file containing the suspected bad blocks. The routine SCAN (in the BADBLK module SCANFILE) searches through the file to determine which blocks are defective.

The SCAN routine tests each block of the file, truncating the trailing blocks from the file. This function occurs in user mode, and retries are inhibited to prevent the disk driver from automatically performing offset recovery. If the block is found to be bad, SCAN uses the MARKBAD truncate option FIB$V_MARKBAD. This option causes the specified blocks (only the last cluster) to be sent to DEALLOCATE_BAD. This operation requires SYSPRV.

DEALLOCATE_BAD, in secondary context, serializes on the BADBLOCK file. A map pointer is added to the last header to map the bad blocks. The end-of-file mark and highwater mark are reset to include these blocks.

In secondary context, SCAN_BADLOG in the F11X module BADSCN is called to scan the pending bad block log and remove any existing BADLOG entries for these blocks. The bad block scanner will also check the BADLOG file for any references to the file when it is done.

When all blocks are truncated from the file, the empty file is deleted and deaccessed.

6.4.3 Switching Stacks

The XQP has an independent operating stack in the impure area that it uses when processing an XQP request. Switching to this private stack ensures efficient and quick exit handling. It also allows synchronization of kernel mode resources. In the case of insufficient resources, the XQP has to wait, or stall, in the mode of the requestor. The normal kernel stack must be emptied before returning, but the XQP private stack allows the call frames on the stack to be saved.

The DISPATCH routine saves the current kernel stack variables in the impure area. The current kernel stack base, contained in the cell CTL$AL_STACK, is written into the first longword of PREV_STKLIM in the XQP impure area. The current stack limit, contained in CTL$AL_STACKLIM, is written into the second longword of PREV_STKLIM. The current frame pointer is saved in PREV_FP in the impure area.

[1] Note that, logically, DSA disks contain no bad blocks. On a DSA disk, bad blocks are be revectored to the RCT when they are written or read. If an error occurs while a block is being read, it is flagged as a "forced error". Rewriting the block is necessary to clear the forced error flag.
Because DSA disks relocate bad blocks when they are rewritten, the bad block scanner never finds the bad blocks again after it rewrites its test pattern. As a result, BADBLK.SYS is always empty on DSA disks.

DISPATCH then sets the first longword of XQP_STKLIM, the base of the private XQP kernel stack, to be CTL$AL_STACK. It also sets the second longward of XQP_STKLIM to be CTL$AL_STACKLIM. XQP_STKLIM also becomes the new stack pointer, which initially points to the base of the private XQP kernel stack.

Below is a list of the pointer updates that occur when the XQP switches from the normal kernel stack to its own private stack. The values in the first column are set to be the values in the second column.

Original Variable	New Variable
CTL$AL_STACK	PREV_STKLIM (first longword)
CTL$AL_STACKLIM	PREV_STKLIM (second longword)
FP	PREV_FP
XQP_STKLIM	CTL$AL_STACK (also SP)
XQP_STKLIM (second longword)	CTL$AL_STACKLIM

Figure 6-24 shows how the XQP switches from the normal kernel stack to the its own internal stack. Note that the stack in this figure grows from bottom to top.

Figure 6-24 Switching from the Kernel Stack to the XQP Internal Stack

```
  Kernel                  Impure Area                XQP Internal
  Stack                                                 Stack

CTL$AL_STACKLIM                                     CTL$AL_STACKLIM
+---------+ ------+  +---------------------+  +----> +---------+
|         |       |  |                     |  |      |         |
|         |       |  |                     |  |      |         |
+---------+       |  +---------------------+  |      |         |
|  call   | ------|-->|       PREV_FP       |  |      |         |
|  frame  |       |  +---------------------+  |      |         |
+---------+  +---|-->|     PREV_STKLIM     |  |      |         |
|         |  |    |  +---               ---+  |      |         |
|         |  |    +-->|      (8 bytes)      |  |      |         |
|         |  |       +---------------------+  |      |         |
|         |  |       |     XQP_STKLIM      |-----+   |         |
|         |  |       +---               ---+  |  |   |         |
|         |  |       |      (8 bytes)      |---+  |   |         |
|         |  |       +---------------------+     |   |         |
|         |  |       |                     |     |   |         |
+---------+ --+      +---------------------+     |--> +---------+
CTL$AL_STACK         |                     |       CTL$AL_STACK
                     |                     |
                     +---------------------+

        ^
        |
growth  |
```

6.4.4 Stalling a Transaction

Because the XQP is multithreaded, an XQP request may be processing, have to stall for a resource wait, and then return to the point of execution. The XQP stalls in the mode of the caller, not kernel mode. The XQP private stack is used to store the context, and ASTs are used to signal that execution may resume.

The XQP is initially entered via AST delivery, so ASTs are blocked while the XQP code is executing (that is, XQP operations are performed at AST level). When the XQP has to stall in the caller's mode for either I/O, a cache wait, or an enqueued lock request, the file system dismisses this kernel AST. A completion AST resumes the thread of execution. XQP activity is generally asynchronous with respect to normal process operation; however, the XQP is itself a serial function.

Two routines in the DISPATCH module are used to accomplish stalls: WAIT_FOR_AST and CONTINUE_THREAD.

If a QIO or ENQ request is queued for which the XQP must stall, the WAIT_FOR_AST routine is called to exit from the current AST so that the completion AST may be delivered. This routine performs the following actions:

- The current frame pointer is saved in XQP_SAVFP in the impure area.
- The XQP channel is made inaccessible by writing a -1 into the CCB$B_AMOD field.
- The previous kernel stack limits and frame pointer are restored.
- A RET instruction is performed to dismiss the AST. Because the frame pointer has been restored, the RET resumes where execution stalled on the original kernel stack.

Performance Monitoring Services (PMS) metering is stopped for the duration of the stall.

The following list shows the pointer updates that occur when the XQP switches from the XQP internal stack to the normal kernel stack.

Original Variable	New Variable
FP	XQP_SAVFP
PREV_STKLIM (first longword)	CTL$AL_STACK
PREV_STKLIM (second longword)	CTL$AL_STACKLIM
PREV_FP	FP

Figure 6-25 illustrates the kernel stack and the XQP internal stack before and after a stall. The process-specific pointers point to the XQP internal stack before the stall and to the normal kernel stack after the stall. Note that the stack in this figure grows from bottom to top.

Figure 6-25 Stalling a Transaction

```
           AFTER STALL                    BEFORE STALL
 Kernel Stack                                   XQP Internal
                                                   Stack
+-------------+ <----+                 +---> +--------------+
|             |      |                 |     |              |
|             |      |                 |     |              |
|             |      +--CTL$AL_STACKLIM--+   |              |
+-------------+ <-+                       +->+--------------+
| EXE$QXQPPKT |   |  +---CTL$AL_STACK----+ | | WAIT_FOR_AST |
|     AST     |   |  |                   | | |  call frame  |
| call frame  |   +--|---frame pointer------+ +--------------+
+-------------+      |                  | | |       0      |
|             |      |  stack pointer------+ +--------------+
|             |      |                   |   |              |
+-------------+      |                  |    +--------------+
|    QIOW     |      |                  |    | Dispatcher   |
+-------------+      |                  |    | call frame   |
|             |      |          ^        |    +--------------+
|             |      |          |        |    |       0      |
+-------------+ <----+   growth |        +---> +--------------+
```

The QIO or ENQ request that was queued specifies the impure pointer (contained in R10) as the AST parameter and the routine CONTINUE_THREAD as the AST routine. CONTINUE_THREAD resets the kernel stack limits to the XQP private stack and restores the saved frame pointer. It then returns to resume execution of the request at the instruction following the WAIT_FOR_AST call.

When the AST is delivered to the CONTINUE_THREAD routine, the following actions occur:

- The impure pointer is restored from the AST parameter.
- The stack limits are set to point back to the XQP stack.
- The saved XQP frame pointer is restored.
- The XQP channel is made accessible again by writing 1 into the CCB$B_AMOD field to indicated a normal kernel mode channel.
- PMS monitoring (including CPU time and number of page faults) resumes.
- A RET instruction is executed, which returns control to the caller of the WAIT_FOR_AST routine (that is, the stalled thread).

Figure 6-26 illustrates the kernel stack and the XQP private stack during and after a stall. The process-specific pointers point to the normal kernel stack before the stall and to the XQP internal stack after the stall. Note that the stack grows from bottom to top.

Figure 6-26 Unstalling a Transaction

```
          BEFORE STARTUP                  AFTER STARTUP

 Kernel Stack                               XQP Internal
                                               Stack
+---------------+ <----+                  +---> +--------------+
|               |      |                  |     |              |
|               |      |                  |     |              |
|               |      +--CTL$AL_STACKLIM--+    |              |
+---------------+ <-+                        +->+--------------+
|AST call frame |   | +---CTL$AL_STACK----+  |  | WAIT_FOR_AST |
|for XQP lock or|   | |                   |  |  |  call frame  |
|QIO completion |   +------frame pointer------+ +--------------+
+---------------+   | |                   |     |      0       |
|               |   +------stack pointer  |     +--------------+
|               |      |                  |     |              |
+---------------+      |                  |     +--------------+
|     QIOW      |      |                  |     |  Dispatcher  |
+---------------+      |                  |     |  call frame  |
|               |      |          ^       |     +--------------+
|               |      |          |       |     |      0       |
+---------------+ <----+   growth |       +---> +--------------+
```

6.5 Error Processing, Status, and Cleanup

One of the basic philosophies of the file system is that it either has to complete an operation successfully or to bugcheck. The routine CLEANUP in the module CLENUP performs the functions necessary to leave file system structures in a more consistent state after a successfully completed file operation.

As a general rule, the file system modules do not clean up after themselves. An operation performed in secondary context must clean up before returning to primary context, but the primary context need not be cleaned up. In primary context, the dispatcher invokes a routine that cleans up before considering the request finished.

Errors can occur at various places while a request processed. Some routines return an error status that is handled by the calling routine. Other file system routines signal errors. When a fatal error is signalled, the dispatcher invokes the ERR_CLEANUP routine, and the error is reported in USER_STATUS.

If ERR_CLEANUP does not initially succeed in leaving file system structures in a consistent state, it is called again. If it succeeds, however, CLEANUP is called. If CLEANUP fails, ERR_CLEANUP is invoked again. This procedure is repeated for a very large, but not infinite, number of times before the file system gives up. ERR_CLEANUP is also resonsible for cleaning up secondary context.

6.5.1 XQP Normal Cleanup

After the XQP has finished successfully processing a request, it must restore the file system structures to their proper state. Normal XQP cleanup involves the following steps:

- Context is changed back to primary if secondary context is current because secondary context is resonsible for performing its own normal cleanup. ERR_CLEANUP resolves secondary context before secondary context is left.
- If the quota file is open for write access, the quota cache is flushed. The VCA$V_CACHEFLUSH bit is set in the quota cache header when an attempt to acquire the quota cache lock fails because the the quota file is write-locked. If the volume has been mounted with the /NOCACHE qualifier, or if it is currently marked for dismount, the buffer caches are flushed.

  All modified buffers are also written to disk, storage map buffers first, in case the storage map is updated before the file headers. No more modified buffers may be created until this request has been completed.
- All windows are invalidated, if requested.
- The directory FCB is deallocated. The FCB is saved if a directory index block is associated with it. If the directory is open for write access, though, directory buffers are discarded, and the directory index block is invalidated.
- The primary FCB is marked stale cluster-wide, if requested. The FCBs are purged unless they are currently accessed or directory index block are associated with them.

6.5.2 XQP Error Handling

When a routine detects an error, it can take one of three actions:

- It can return the error as a return status.
- It can store the error status in the user return status cell (USER_STATUS) by calling the ERR_STATUS macro. USER_STATUS is a two-longword vector that is returned to the user in the IRP$L_MEDIA field. These two longwords form the IOSB returned to the user. ERR_STATUS only stores the status value if the existing value is either success or informational. This action is taken for errors that are not fatal but that the user should see. Because invoking ERR_STATUS writes USER_STATUS directly, calling routines cannot intercept the error.

- It can invoke the ERR_EXIT macro. This macro signals the condition value by performing a CHMU instruction of the argument, which declares an exception to VMS. If a condition handler is present, it will deal with the condition. Otherwise, the macro performs a return instruction with the value left in R0.

  The error is reported in USER_STATUS. The DISPATCHER condition handler MAIN_HANDLER copies the argument into USER_STATUS (unless USER_STATUS already indicates an error), places USER_STATUS into the value that will be restored into R0, and unwinds to the routine that established the handler. The mainline call to an XQP processing routine returns with the status value passed to ERR_EXIT, and the processing routine is aborted. No XQP routines handle the unwind condition.

6.5.3 Event Notification

The file system provides two sets of messages. A privileged user may request notification of interesting file system events. The system itself requests notification of security-related events. These two sets of events are reported in the following way:

- The SET WATCH command allows a suitably privileged user to request notification of significant events in the file system. The list of significant events is stored as bits in the array PIO$GW_DFPROT, which is indexed by the XQP event index. Various routines in the file system check their corresponding bit and invoke the NOTIFY_USER routine to send the user a message.
- When all file system activity has been completed for a request, the PERFORM_AUDIT routine is called, if necessary. During the course of the request, audit blocks were placed by CHECK_PROTECT in the impure cell AUDIT_ARGLIST. These requests are passed to NSA$EVENT_AUDIT one at a time.

  For each audit entry, the specified file ID from the supplied header must be translated to a full file specification. As a result, performing an audit is deferred until the request has been processed because the FID_TO_SPEC routine seriously affects other file system operations; it releases the primary serialization lock. The one exception is that a WRITE_AUDIT call appears in the MARK_DELETE routine because the file will no exist to be audited after MARK_DELETE operates.

6.6 Termination of Processing

After all pending requests have been processed and the necessary cleanup has been performed, DISPATCHER calls the UNLOCK_XQP routine to release the XQP synchronization locks. The serialization lock is released, the value block is updated, the current volume allocation lock (if any) is released, the in-process buffer credits are returned to the buffer pool, PMS monitoring is halted, and the cache interlock is released. DISPATCHER then calls the routine IO_DONE.

IO_DONE posts I/O completion for the file system request. It performs the following actions:

- Moves USER_STATUS into IRP$L_MEDIA (which is actually a quadword).
- Decrements the transaction count for the VCB.
- Clears the name string descriptor length in the complex buffer packet to prevent the name from being written back for efficiency.
- Copies the local FIB back into the complex buffer packet. If attributes have been changed (for instance, a Rename operation was performed), the most recent copy of the FIB needs to be preserved.
- Sets IRP$L_BCNT to ABD$C_ATTRIB for non-read functions so that the attributes don't get written back to the user's buffer. However, if the READ bit (IRP$V_FUNC in the IRP$W_STS field) is set, the attribute text is kept.
- Calls the CHECK_DISMOUNT routine.

The CHECK_DISMOUNT routine, in the CHKDMO module, performs deferred dismount processing. The UCB linked list for the volume or volume set is traversed, and any volume is dismounted whose DEV$V_DMT bit is set (which indicates that the volume is marked for dismount) and whose transaction count is 1 (which indicates that the volume is idle, except for the current process.)

During a volume dismount, the following steps are taken:

- The UCB$V_DISMOUNT bit is set while the I/O database is locked to prevent other processes from starting I/O on the volume.
- An IO$_UNLOAD/IO$_AVAILABLE function is issued.
- If the volume is mounted cluster-wide, the value block for the volume lock is obtained in PW mode.
- The high bit of the UCB$W_DIRSEQ field is cleared to warn RMS of the volume dismount (for more information, refer to the Section on the RMS directory cache).
- The UCB$W_REFC field is decremented.
- The AQB$W_MNTCNT field is decremented and if it is zero, it is removed from the AQB list.

- All FCBs, ACBs, and WCBs are deallocated, and the access locks are dequeued by forcing FCB$W_REFCNT to 0.
- The FID and extent cache locks are dequeued, and the caches are deallocated.
- The quota cache lock is dequeued, and the quota cache is deallocated.
- The volume lock (VCB$L_VOLLKID) is dequeued.
- The shadow lock is dequeued.
- For volume sets, the RVT list entry is cleared, and the RVT$W_REFC field is decremented. If it is zero, the structure lock and the blocking lock are dequeued. BLOCK_CHECK is cleared so DISPATCHER will not release the block lock, and the RVT is deallocated.
- For single volumes, the blocking lock is dequeued.
- The VCB is deallocated.
- Demote the volume lock, if any, either to CR mode if the volume is not allocated or to EX if it is. The value block is cleared if this is the final dismount.
- The routine IOC$DALLOC_DMT in the SYS module IOSUBPAGD is called to deallocate the volume.
- The buffer cache and the AQB are deallocated.

CHECK_DISMOUNT then returns to IO_DONE to finish posting I/O completion for the request.

The I/O postprocessing routines (in the SYS module IOCIOPOST) are chiefly responsible for accumulating the total number of bytes transferred in the I/O request and for starting further I/O processing if the request has not been completed. When the IRP$V_VIRTUAL bit in the IRP$W_STS field is set, the XQP adds the number of bytes transferred (contained in IRP$L_BCNT) to the number of bytes accumulated so far (contained in IRP$L_ABCNT). The value in IRP$L_ABCNT is then compared to the value in IRP$L_OBCNT, which gives the original byte count. When the two values are equal, the request is complete.

Different routines in IOCIOPOST are called depending on whether the request need to be completed in process or system context. Virtual I/O requests are completed in process context, and so the IOC$BUFPOST routine is called.

Device I/O requests are completed in system context, and so the routine IOPOST is called.

6.6.1 Virtual I/O

Because the XQP performs virtual file system functions within process context, issuing an IOPOST software interrupt and a special kernel AST to post I/O completion is unnecessary. As a result, IO_DONE optimizes the code by calling (via JSB) the special entry point IOC$BUFPOST in IOCIOPOST.

IOC$BUFPOST executes the same code executed by the IOPOST software interrupt; PCB quotas are reset, and the equivalent of the special kernel mode AST completion routine is set up, which specifies one of the two following routines:

- BUFPOST for XQP functions (except window turns) requiring a complex buffer and buffer I/O.
- DIRPOST for direct I/O.

If, however, the IRP$L_PID field is negative, which indicates a system address, special I/O postprocessing must be performed, and an I/O post interrupt must be signalled.

After returning, IO_DONE posts an event flag, and then another JSB instruction executes the special kernel AST code to complete posting of the I/O completion. If the specified completion routine was BUFPOST, the IRP-described buffers (such as the FIB) are copied back to the user buffers, the accumulated buffered I/O count in PHD$L_BIOCNT is incremented, the complex buffer is deallocated, and DIRPOST is called.

DIRPOST performs the following general I/O completion activities:

- Updates process header quotas (PHD$L_DIOCNT is incremented).
- Decrements the channel activity count in CCB$W_IOC, showing that there is no more I/O in progress.
- Sends a deaccess request to the XQP if the activity count is zero and CCB$L_DIRP indicates a pending deaccess operation (CCB$L_DIRP contains a non-zero value).
- Writes the user IOSB.
- Sets the event flag specified in the $QIO call by calling SCH$POSTEF.
- Queues the user AST by using the IRP as an ACB.
- Deallocates the I/O packet or the IRPE.

Finally, the DISPATCHER routine calls FINISH_REQUEST. This routine sets IPL to IPL$_SYNCH and lowers the volume activity count by decrementing the value in the VCB$L_ACTIVITY field by 2. For a volume set, the activity count is decremented in the RVT$L_ACTIVITY field of each volume. FINISH_REQUEST then resets IPL to 0 and returns to the DISPATCH routine—the routine to which the original AST was delivered.

DISPATCH restores the original kernel stack limits and frame pointer, decrements the PCB$B_DPC field to allow process deletion and suspension again, makes the XQP channel inaccesible, and returns.

Figure 6-27 illustrates the kernel stack and the XQP private stack after the request has been completed. The process-specific pointers are reset from the XQP internal stack to the normal kernel stack. Note that the stack grows from bottom to top.

Figure 6-27 XQP Transaction Completion

```
                     AFTER                     BEFORE

 Kernel Stack                                     XQP Internal
                                                     Stack
+---------------+ <----+                   +--> +------------+
|               |      |                   |    |            |
|               |      |                   |    |            |
|               |      +--CTL$AL_STACKLIM--+    |            |
+---------------+ <-+                           |            |
|EXE$QXQPPKT AST|   |  +---CTL$AL_STACK----+    |            |
| or completion |   |  |                   |    |            |
|AST call frame |   +--|---frame pointer   |    |            |
+---------------+      |                   |    |            |
|               |      |   stack pointer---+    |            |
+---------------+      |                   |    +------------+
|     QIOW      |      |                   |    | Dispatcher |
+---------------+      |                   |    | call frame |
|               |      |          ^        |    +------------+
|               |      |          |        |    |     0      |
+---------------+ <----+   growth |        +--> +------------+
```

6.6.2 Device I/O

Because the XQP executes within process context, it does not have to issue an IOPOST software interrupt and a special kernel AST to post I/O completion. However, when a device driver or FDT routine posts I/O completion, it calls a routine (IOC$REQCOM) that inserts the IRP at the tail of the I/O postprocessing queue (located by the global cell IOC$GL_PSBL) and requests a software interrupt at IPL$_IOPOST (IPL 4).

The routine IOPOST in the SYS module IOCIOPOST executes as a result of the I/O posting interrupt. All driver I/O is completed there. It removes I/O packets from the postprocessing queue (located by the global cell IOC$GL_PSFL) and processes them until completion. IOPOST also performs the following actions:

- Unlocks any system memory used for the I/O request.

- Increases process quota usage by incrementing the PCB$W_BIOCNT or PCB$W_DIOCNT fields.
- Unlocks the user's pages if the request was a direct I/O (indicated by the bits in the IRP$W_STS field).
- Deallocates the buffer if the request was a buffered write.
- Transfers the information from the buffer to the user's part of the addresss space if the I/O was a buffered read.
- Posts the I/O status to the user's I/O status block.

However, because a driver or FDT routine does not execute in process context, a special kernel AST is queued to the process that initiated the I/O request. The I/O packet is turned into an AST control block and placed into the AST queue for the process that reqeusted the I/O. The kernel AST routine address is set up to be a part of the IOPOST code. The IOPOST interrupt service routine then loops back to remove another I/O packet from the beginning of the post queue (located through global pointer IOC$GL_PSFL). When the queue is empty, the IPL 4 software interrupt is dismissed.