VMS File System Internals

Kirby McCoy

digital

DIGITAL PRESS

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

Order number EY-F575E-DP

This book was produced by Digital's Corporate User Publications Group with the VAX DOCUMENT electronic publishing system.

Library of Congress Cataloging-in-Publication Data
McCoy, Kirby
VMS File System Internals/Kirby McCoy.
p. cm.
Includes index.
ISBN 1-55558-056-4
1. VAX/VMS (Computer operating system) 2. File organization (Computer science) I. Title.
QA76.76.O63M385 1990
005.74-dc20 90-3746
CIP

Lovingly dedicated to the memory of my father,
C. C. McCoy, Sr.

BNA 2/6/91

Contents

3 Volume Structure Processing

4 Cache Processing on a Single Node

5 The ACP Functions

6 The XQP and I/O Processing

7 Serialization of File System Activity

Preface

VMS File System Internals provides information about the internal components of the VMS Version 5.2 file system, which is that part of the VAX/VMS operating system responsible for storing and managing information and files in memory and on secondary storage.

Intended Audience

This book is intended primarily for software specialists, system programmers, and other users who wish to understand the underlying components of the VMS file system.

System managers may benefit from understanding the details of file system data structures and caches when they configure the system. Application designers may likewise benefit from understanding the file system structures and logic that may affect various design decisions.

The audience is assumed to be familiar with the VAX architecture, the VMS operating system as a whole, I/O devices, and device drivers.

Document Structure

This book contains the following eight chapters:

- Chapter 1 introduces the VMS file system. It provides insight into how the file system has evolved, and gives an overview to the file system user interface.
- Chapter 2 discusses the Files–11 On-Disk Structure, including the basic structures of the VMS file system and general file system concepts.
- Chapter 3 covers volume structure processing. Major topics include the Initialize, Mount, and Dismount Utilities, and the structures of the I/O database.

- Chapter 4 explains the fundamentals of cache processing on a single node. This discussion includes cache structures, special caches, and basic caching algorithms.
- Chapter 5 discusses major and miscellaneous ACP functions, including access, create, delete, modify, deaccess, and ACP control. It also gives an overview to the Queue I/O (QIO) interface.
- Chapter 6 provides a detailed discussion of the XQP and I/O processing, including an in-depth explanation of the the QIO interface. Topics also include XQP dispatching, XQP code execution, and I/O postprocessing.
- Chapter 7 describes the various serialization techniques used to synchronize file system activity, including the distributed lock manager, raised IPL, and the file system structures themselves.
- Chapter 8 discusses the file system in a VAXcluster environment. It covers the coordination of clusterwide file system structures and resources, and how file system requests are passed to all nodes of a VAXcluster.

Acknowledgments

This project has been both long and hard, and I am thankful for the caliber of help that I have had.

My first thanks go to Susan Denham and to Ann MacDonald, my current and my former supervisor, for supporting me in what at times has seemed a neverending endeavor. Both of these women were deeply committed to ensuring that I succeed in writing this book.

I am grateful to all the people who provided editing and production support. Lisl Urban edited the first versions of all the chapters. Judy Blachek provided a comprehensive edit of the final version of the whole book out of the goodness of her heart. Jill Angel magnificently coordinated the entire production. Lee Segal, Chris Caron, Sheila Lawner and Brenda Rogers of Graphic Services provided book design and production assistance. Paul King provided the art work, which is particularly notable because it was given to him at the last minute.

I also thank Chase Duffy and Michael Meehan of Digital Press for their publishing experience and their on-going support.

I am grateful for the help and suggestions provided by Ruth Goldenberg and the rest of the VAXworks team. I also acknowledge the help provided by friends and reviewers (you know who you are!) around the world, whom I never would have met otherwise.

I also thank the many talented VMS engineers who reviewed and significantly improved portions of the book: Paul Destefano, Paul Houlihan, Hai Huang, Mark Pilant, and Ralph Weber. I appreciate all the errors and omissions they found.

I also appreciate the help of Christian D. Saether, designer and implementer of the XQP. He provided a document on XQP internals that later became the core of Chapters 4, 6, and 7. He also helped establish the outline for the book and provided reviews of some of the early chapters.

I am especially grateful to two brilliant engineers, without whom I would not have been able to write this book. It has indeed been an honor and a privilege to have been able to work with both of them.

Andy Goldstein was the original perpetrator of ODS–2, and he wrote a document on the Files–11 On-Disk Structure that was later subsumed into Chapter 2. Notoriously overworked, he nevertheless patiently answered a multitude of questions, and took the time to provide a comprehensive, incisive, and generally excellent review of the whole book, enlivened by his barbed sense of humor.

Keith Walls is the current architect and maintainer of the file system, and I am deeply indebted to all the extra work he put in to make sure the book was successful. This help included (but was not limited to) giving weekly tutorials and code walk-throughs, answering innumerable questions, providing the original version of Chapter 1, bringing a pizza to work at night so we could review a chapter without missing dinner, bringing comments in at midnight and going over them with me till the wee hours when I was under a deadline, and generally helping me put the mysterious pieces of the file system together. Keith kept me (and the book) going when the project stalled, and provided help of a nature that went far above and beyond the call of duty.

Finally, I would like to thank all those who have designed, implemented, or contributed to the VMS file system. This book is but a small tribute to their monumental engineering effort.

Kirby McCoy
April 1990

Conventions

The following conventions are used in this manual:

. . .	In examples, a horizontal ellipsis indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
<>	In format descriptions, angle brackets indicate that the user must supply the information.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason.
UPPERCASE TEXT	Uppercase letters indicate that you must enter a command (for example, enter OPEN/READ), or they indicate the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.
-	Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows.
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.
data structures	All data structures are assumed to run right to left. That is, the lowest addressed byte (or bit) in a longword is on the right-hand side of a figure, and the most significant byte (or bit) is on the left-hand side.
system parameter	This term is used to describe any of the adjustable parameters (also called SYSGEN parameters) that are used to configure the system.
Executive	This term refers to those parts of the operating system that reside in system virtual address space.

Chapter 1

Introduction to the VMS File System

Begin at the beginning . . . and go on till you come to the end: then stop.
Lewis Carroll

If you want to fix something, you are first obliged to understand, in detail, the whole system.
Lewis Thomas

Outline

1.1 Introduction

The file system of the VAX/VMS operating system has an interesting history and has evolved to occupy an interesting and unique place in the computing industry. This book is intended to deliver a detailed description of the VMS file system for Version 5.0 of VAX/VMS. Particular attention is given to the methodology and tactics employed in the file system to achieve the synchronization, efficiency, and flexibility it does.

1.2 Tasks of a File System

File systems in general are exceptional pieces of code within operating systems. They are relied upon to deliver data with absolute integrity and virtual immunity to security problems and media defects. Furthermore, file systems are expected to do this with minimal impact on system performance. For example, the XQP runs as a kernel-mode asynchronous system trap (AST) delivered to the requesting (and owning) process. As a result, the processing performed by the XQP is absolute overhead, and is measured as such in all forms of monitoring.

A timesharing file system is, by its very nature, unique in the computer science realm. Its implementation should provide the following:

- The imposition of a hierarchical name space on an otherwise flat logical space
- An immutable name space
- Multiplexed usage of block space
- Implicit synchronization
- A bridge between the unprivileged user and the privileged operating system
- File management on secondary storage
- Private and secure storage
- The sharing of files in a controlled fashion
- A fault-free environment for the upper layers and to the operating system

  A failure (such as a security violation or the detection and repair of a bad block) is either hidden from upper levels (in the case of bad block handling), or it is returned to the user as an error (in the case of the security violation). However, the latter case actually represents a success because the file system successfully detected an attempted security violation and prevented it.

Most importantly, the file system is expected to perform all these tasks without making a significant impact on the environment or the users whom it serves.

1.2.1 Evolution of the VMS File System

The VAX/VMS operating system evolved from the RSX–11M operating system in the 1970s. On the RSX series of operating systems, the **Files–11 On-Disk Structure Level 1** (ODS–1) file system functions were implemented as a separate process called an **ancilliary control process** (ACP). Thus, each request that involved the file system ACP likewise involved a process context switch to the ACP, as well as a context switch back to the requesting process.

Early versions of VAX/VMS also used a file system ACP to handle file I/O functions. Although VAX/VMS provided a more secure domain than that provided by RSX and, in Version 1.0, tightly bound volume sets under the heading of ODS–2, these early implementations of the VMS file system still required the context to be switched and data to be transferred between processes in order to perform file system operations.

In addition to the overhead incurred by a separate process context, the ACP design also suffered from being a systemwide bottleneck because all process requests were funneled through a single ACP for any given disk. Despite the apparent shortcomings of these early versions of the VMS file system implementation, synchronization was implicit within the ACP. Updates and privileged references to the file system in-memory data structures were synchronized by processor interrupt priority level (IPL), but only one VMS process had control of any given disk. In other words, all file-system-related requests were channeled through an entity of atomic or implicit synchronization.

1.2.2 Creation of the XQP

In VAX/VMS Version 4.0, Digital introduced the VAXcluster system. In order to maintain rational and useful file sharing within the cluster, the file system had to adopt a different synchronization technique because processes could no longer depend on the implicit synchronization offered by the scheduling entity (the ACP). So it became necessary to develop a means by which many "ACPs" could synchronize across the boundaries within the VAXcluster.

As an answer to the problems posed by the now-outdated ACP, the **extended QIO processor** (XQP) was designed. It was intended to perform the following tasks:

- Eliminate the ACP bottleneck
- Provide synchronization between processes within a VAXcluster
- Ensure that copies of data structures across the VAXcluster were properly maintained (generally by an invalidation-and-update technique)

On the basis of these requirements, the XQP was developed as a "per-process ACP." In this way, each process mapped the code of the XQP from a global P1-only image section and maintained per-process, private, in-memory data structures relevant to the file system operations of that process. Thus, the file-system activity of any one process had no impact on that of any other process on the system, except where file (read and write) sharing was concerned.

The synchronization between processes became solely dependent on the distributed lock manager, which allowed process-based XQPs to express interest at a file granularity level as opposed to the volume granularity of the ACP system. In-memory data structures were still protected by the IPL scheme offered by VMS (and later by the spin lock mechanism implemented for symmetric multiprocessing).

So where the ACP was a separate process, the XQP was merged into the process (so that only the impure area was charged to the working set). The ACP handled requests for all users; the XQP handled requests for the single user of its process. The ACP was usually single threaded and lacked cluster synchronization, but the XQP effectively provided multiple copies and cluster synchronization.

1.2.3 VMS File System in a VAXcluster

The VAX/VMS file system is further distinguished in its involvement in the Digital VAXcluster technology. Users of VAXclusters have long enjoyed transparent synchronization and transparent file-sharing by the cluster-integrated VMS file system.

In fact, the development of the VMS operating system itself is now fully dependent on its own VAXcluster technology and the accompanying file system. In 1977, compiling a new VAX/VMS operating system from its source files consumed the resources of an entire VAX–11/780 for 12 hours. In 1979, building the VMS operating system consumed the resources of an entire VAX–11/782, with nearly twice the horsepower, for the same amount of time.

Today, a complete build of the VAX/VMS operating system consumes the resources of an entire VAXcluster consisting of two VAX 8800-class machines, three VAX 6000-class machines, and miscellaneous VUPs for still the same amount of time. If it were not for VAXcluster technology, building VMS would probably be a very lengthy process. Further, if all file activities went through a single-threaded ACP, more time would be spent waiting for the ACP than for the operation to complete.

1.2.3.1 VAX/VMS Environment and the File System

The VMS file system stands out in many ways. Unlike many operating system file systems, the VMS file system was designed as an integral component of the operating system. In the architectural phase of VAX/VMS hardware and software development, a successful file system was considered essential to the success of the hardware-software architectural teams.

For example, data reliability features in the VAX architecture, such as the CRC (Cyclic Redundancy Check) instruction, were used to advantage in the VMS file system and related utilities. In addition, the memory-management unit (the page) is the same size as the file system block. I/O postprocessing performed by the operating system in response to I/O completion consists of explicit considerations of file system I/O.

In addition, the page fault handler of VMS deals directly with files and file blocks (or pages). In fact, the VMS page-fault mechanism provides the means by which image pages are read from disk images (files) into memory for execution.

Many parts of the VMS operating system have knowledge of the VMS file system. The swapper, although it does not have P0 or P1 space (and therefore is the only VMS process that cannot map the XQP code) is involved in file-related I/O and in file system synchronization.

The XQP has significantly influenced the VMS operating system. Without the XQP, system-owned locks would not be necessary in the distributed lock manager. The major premise of the distributed lock manager is that any lock has an owner process. In the case of the file system, however, it is necessary for some locks to outlive the process that created them (for example, a disk that has been mounted clusterwide).

1.3 User Interface to the File System

Perhaps the most distinguishing feature of the VMS file system is that it makes every attempt to detach itself from all the I/O operations it can. This is a benefit inherited from the ACP design of the RSX and VMS file systems from which it evolved. The VMS file system is involved only with I/O that requires access to the file system metadata. Once the file system has been called to make an access path to a file, it represents its presence, as far as it can reasonably do, by leaving nonpaged data structures (window control block, file control blocks) that provide a mapping from virtual (file-based) blocks to logical (disk-based) blocks.

In this way, the file system only needs to be involved in the initial open operation (and associated synchronization) and in window turning (updating the virtual-to-logical map) to provide the user with direct access to the blocks of the file through the window control block. In this sense, the VMS file system is integrated completely into the QIO subsystem of the operating system.

In fact, all I/O to a file can be expressed in terms of virtual QIOs, which allows a record management subsystem (like RMS or Rdb) to impose its own structure within files without the involvement of the file system. Despite the level of integration between the operating system and its file system, the XQP presents an object-oriented and hierarchical layer upon which upper-level facilities can build.

In addition to these constraints and design decisions presented by the VMS file system, I/O passed to the XQP can be made completely asynchronous to the upper layers, and in a transparent manner to the application designer or programmer.

1.3.1 VMS I/O System

The VAX/VMS I/O system is composed of several layers. The top layer is the VAX Record Management Services (RMS), which provides controlled access to files and records. All VAX high level languages invoke VAX RMS to perform I/O. VAX RMS is also the recommended I/O mechanism for the assembly language programmer.

The middle layer is the Queue I/O ($QIO) system service, which interfaces with the XQP to perform device-dependent I/O. A programmer would use the $QIO system service when accessing devices not supported by RMS; when performing I/O operations not supported by RMS; or when performing I/O operations not supported by the language's interface to RMS.

The bottom layer is the device driver itself. The $QIO service acts as the interface to the device driver, which is rarely accessed directly by the application programmer.

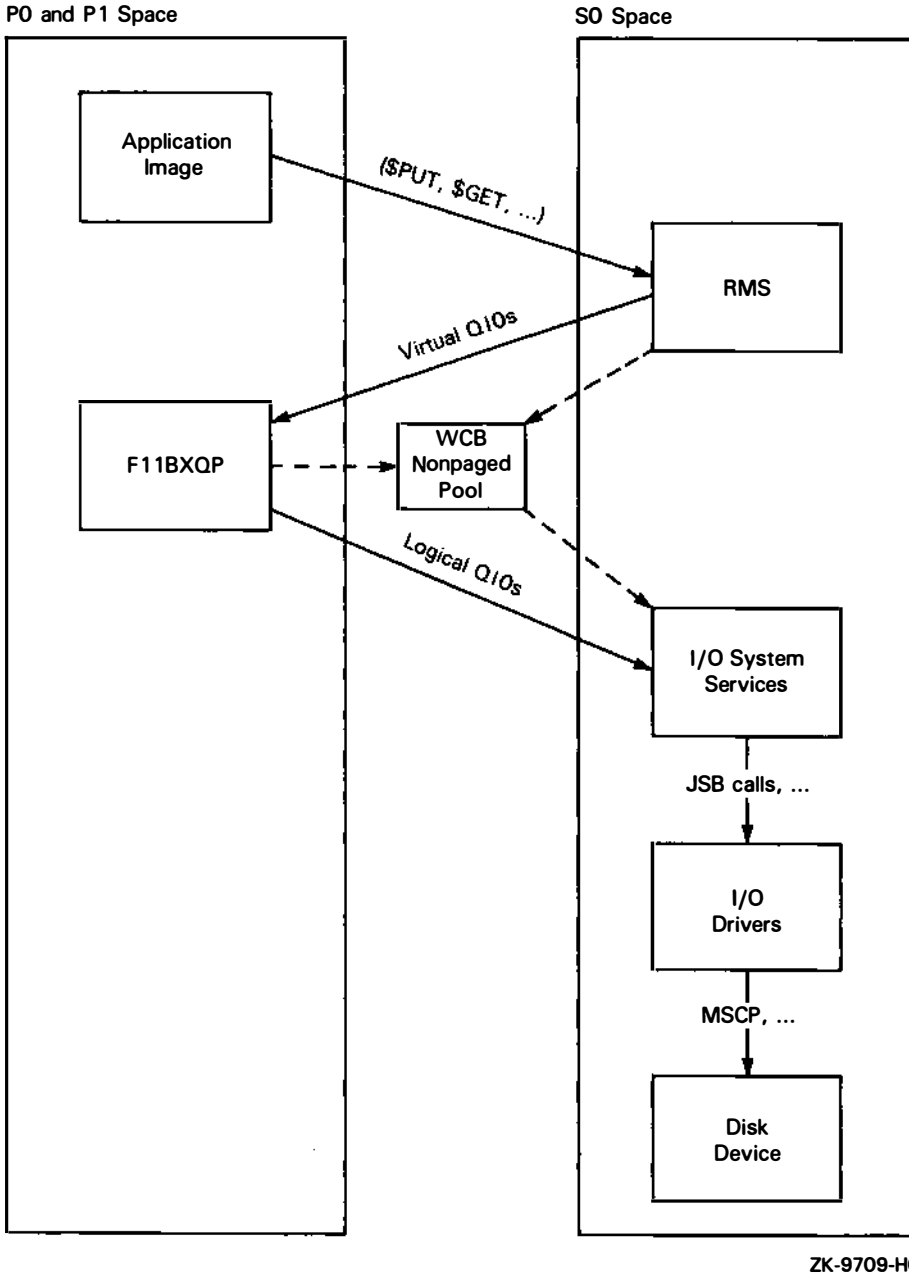
A user program can interface with the I/O system at different levels, depending on its requirements. At each level, the user program makes tradeoffs between ease of use and execution speed. As a general rule, the lower the level at which the user program interfaces with the VAX/VMS executive, the less overhead is involved in the I/O operation. On the other hand, less opportunity is provided for data caching.

In most instances, a programmer uses VAX RMS either directly or implicitly to perform input and output operations to file-structured devices. Access to real-time devices is usually done by directly invoking the $QIO system service to the driver level.

Any functions that can be performed on a device can also be enqueued with an appropriate $QIO.

Figure 1–1 shows the relationship between the components of the I/O system.

Figure 1–1: The Components of the I/O System



1.3.2 Queue I/O Request Service

The Queue I/O request service ($QIO) queues an I/O request to a channel associated with a device. The $QIO service completes asynchronously; that is, it returns to the caller immediately after queuing the I/O request, without waiting for the I/O operation to complete.

$QIO operates only on assigned I/O channels and only from access modes that are equal to or more privileged than the access mode from which the original channel assignment was made.

$QIO consumes the process quota for the following resources:

- Buffered I/O limit (BIOLM) or direct I/O limit (DIOLM)
- Buffered I/O byte count (BYTLM)
- AST limit (ASTLM), if an AST service routine is specified

System dynamic memory is also required to hold a database to queue the I/O request, and additional memory may be required on a device-dependent basis.

Chapter 2

Files–11 On-Disk Structure

Within that awful volume lies
The mystery, of mysteries!
Sir Walter Scott

Something deeply hidden had to be behind things.
Albert Einstein

Outline

2.1 Introduction

A major component of the VMS operating system is the **file system**. The file system, also called the **file control processor** (FCP), maintains the structure and integrity of data stored on file-structured devices such as disks.[1]

The file system is responsible for the following tasks:

- Maintaining the directory files on the volume
- Opening, closing, creating, deleting, extending, and truncating files
- Managing free space on the volume
- Ensuring the integrity of files
- Mapping logical blocks to virtual blocks
- Translating RMS data requests for device drivers

The standard file structure for all medium-to-large PDP–11 and VAX systems is **Files–11**. This book, and this chapter in particular, describes the **Files–11 On-Disk Structure Level 2** (ODS-2) used by VAX/VMS systems. The following sections provide a conceptual overview of the basic components of the file system—volumes, files, and directories.

2.2 Basic Concept of a Volume

A **volume** is the basic medium with a Files–11 structure. It is an ordered set of **logical blocks**. A logical block is an array of 512 8-bit bytes. If the volume contains n logical blocks, the logical blocks are consecutively numbered from 0 to $n - 1$. The number assigned to a logical block is called its **logical block number** or **LBN**.

In practice, a volume should be at least 100 blocks to be useful, and Files–11 can describe volumes up to 2^{32} blocks.

The logical blocks of a Files–11 volume must be randomly addressable. The volume must also allow transfers of any length up to 65,536 bytes in multiples of four bytes. If the data is longer than 512 bytes, consecutively numbered logical blocks are transferred until no more data remains to be transferred.

In other words, the volume can be viewed as a partitioned array of bytes. It must allow read and write operations on arrays of any length up to 65,536 bytes, provided that the array starts on a logical block boundary and that its length is a multiple of four bytes. When only part of a block is written, the contents of the remainder of that logical block are undefined.

[1] Block addressable storage devices such as disks and TU58 magnetic tapes are the assumed media, and they are generically referred to as disks.

The logical blocks of a volume are grouped into **clusters**. The cluster is the basic unit of space allocation on the volume. Each cluster contains one or more logical blocks, and the number of blocks in a cluster is called the **volume cluster factor** or the **storage bitmap cluster factor**.

2.2.1 Volume Identification

The file system identifies a volume as a Files–11 volume by its **home block**. The home block is located at a defined physical location on the volume—usually LBN 1. The file system verifies the home block by its checksums and predictable values. The home block also contains a **volume label**, which is an ASCII character string, to identify the volume.

The home block also serves another important function. It contains a pointer to the index file INDEXF.SYS, which is the file that contains the information the file system needs to access the rest of the files on the volume.

For more information on the home block, refer to Section 2.5.1.2.

2.2.2 Volume Integrity

One of the basic concepts of the file system is that it be robust, or tolerant of system failure, particularly across a VAXcluster. The file system must be able to tolerate the random failure of a node in the VAXcluster without destroying file data or access to files from other nodes in the cluster. In other words, a critical requirement of a robust file system is the integrity of a volume. It is imperative that the data on a volume be correct and valid at any given time.

The essential way in which the integrity of a volume is ensured is the redundancy of key structures on the volume. For example, multiple copies of the home block along the home block search sequence allow access to the volume even if the primary home block is corrupted. For more information on the home block, see Section 2.5.1.2.

Another structure that is recorded multiple times on the volume is the index file header. The backup index file header allows data on the volume to be recovered even if the primary index file header is corrupted.

A second method that ensures volume integrity is the order in which specific structures are updated so that a system failure in the middle of an operation does not compromise the entire volume. For example, to extend a file, the file system must allocate free storage from the disk. The desired number of blocks is first reserved in the storage bitmap. Only then does the file system write out to disk the file header of the file being extended, which associates those blocks with that particular file.

If the header were written out to disk before the blocks were reserved in the storage bitmap, two files could potentially map the same blocks, resulting in multiply allocated blocks and file corruption. So any structure that involves multiple disk blocks must be sensitive to the order in which they are written.

Yet another way that volume integrity is ensured is the distinction between directory structure and file structure. This difference allows files to be recovered from a directory that has been deleted or corrupted.

2.2.3 Volume Sets

A collection of related disks that is treated as one logical device is called a **volume set**. Although each volume contains its own Files–11 structure, there is only one directory structure on the volume set. Files on the volumes of the set are referenced with a **relative volume number**, which uniquely determines the disk in the set on which the file is located.

2.2.3.1 Tightly Coupled Volume Sets

A volume set that is consistent and self-identifying is called a a **tightly coupled volume set**. The volume label of each volume in the set is unique within the set and is different from the **structure name**, which is a string of up to twelve ASCII characters which identifies the volume set. Relative volume 1 (the root volume) of the set contains a file (VOLSET.SYS) that lists the volume labels of all the volumes in the set and thus associates volume labels with relative volume numbers. Each volume is identified as part of the set by its structure name, volume label, and relative volume number.

For more information on VOLSET.SYS, refer to Section 2.5.6.

2.2.3.2 Loosely Coupled Volume Sets

A **loosely coupled volume set** is a collection of volumes that is not self-identifying. It does not contain a file that lists the volume labels. Moreover, only one file per volume may cross from one volume in the set to the next, and files that cross volumes may only be processed sequentially.

Loosely coupled volume sets emulate multivolume magnetic tape, which allows a file to be sequentially read or written with only one volume mounted at a time.

Correct sequencing of the volumes that hold a particular file is the responsibility of both the system operator and the application using the volume set, although there are checks and corrections that can catch most handling errors.

The Backup Utility (BACKUP) produces a loosely coupled volume set when, for instance, a large disk such as an RA81 is backed up onto multiple RA60s.

2.3 Basic Concept of a File

A **file** is an organized collection of data. Files contain any data on a volume or volume set that is of interest (that is, all the blocks that are not currently available for allocation).

To be independent of disk drivers, the file system imposes a logical structure on the data on each disk. Essentially, the FCP treats a disk as a logically contiguous series of data units called **logical blocks**. A logical block contains 512 8-bit bytes and is numbered from 0 to $n - 1$, where n is the number of blocks on the disk.

2.3.1 Logical to Virtual Mapping

A file is an ordered set of **virtual blocks**, where a virtual block, like a logical block, is an array of 512 8-bit bytes. The file system regards a file as being virtually contiguous.

Unlike logical blocks, however, the virtual blocks of a file are consecutively numbered from 1 to n, where n is the highest numbered block that has been allocated to the file. A logical block and a virtual block describe the same physical unit of storage on the disk; the only difference between them is the way they are numbered.

Logical blocks have logical block numbers (LBNs), and virtual blocks have virtual block numbers (VBNs). Virtual blocks have LBNs, but logical blocks do not have VBNs, unless they are allocated to a file.

Virtually contiguous does not necessarily mean logically contiguous. There is more than one file on a disk. As these files consume space on the disk, there are fewer available contiguous logical blocks. Eventually, the file system creates or extends a file so that portions of it reside on different parts of the disk. The blocks retain their serial VBNs, but they are no longer logically contiguous.

Files–11, which is device-independent, is responsible for associating, or mapping, virtual blocks to unique logical blocks in a volume set. For example, if a user requests access to a block within a file, the file system calculates the logical block on the volume that corresponds to the VBN. It takes into account the fact that the virtual blocks may or may not be logically contiguous.

After calculating the LBN, the file system requests that block from the disk driver for the device containing the file. The driver then translates that request into the cylinder/track/sector, or **physical block,** that the device hardware must read or write.

Files can be either dense or sparse. A file in which all the VBNs less than or equal to the highest allocated VBN have been mapped to a corresponding LBN in the volume set is a **dense** file.

On the other hand, files that are **sparse** contain virtual blocks that have not been allocated logical blocks. Unallocated virtual blocks are represented by mapping data that contains an LBN of all 1s. Sparsely allocated files are not currently supported.

2.3.2 File Identification

Every file in a volume set is uniquely described by a number called a **file identifier**, a **file ID**, or simply a **FID**. A file ID is a 48-bit binary value that is supplied by the file system when the file is created. Users must supply it when they want to access a particular file. The file identifier points to the location of the file header, which contains a listing of the extent or extents that locate the actual data on the disk.

The file ID is divided into four fields:

- File number
- File sequence number
- Relative volume number
- File number extension

These fields are shown in Figure 2–1 and are described in Table 2–1.

Figure 2–1: Format of the File Identifier

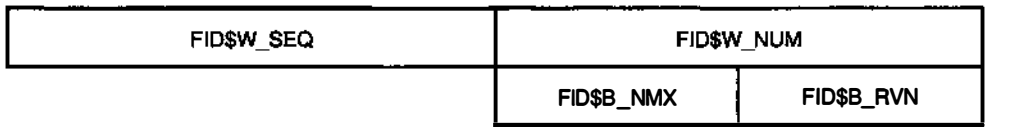


Table 2–1: Contents of the File Identifier

Field Name	Description
FID$W_NUM	File number. This field contains the low 16 bits of the file number. With the FID$B_NMX field, it forms a 24-bit file number that locates the file within a particular volume of the volume set. The set of file numbers on a volume is not totally dense. The file number uniquely identifies one file on that volume at any one time. File numbers for a volume start with the number 1; 0 is not valid. File numbers with zeros in the low 16 bits (multiples of 65,536) are not used.
FID$W_SEQ	File sequence number. It represents the current use of a particular file number on a volume. When a file is deleted, its file number can be used again. Each time a file number is reused, a different file sequence number is assigned to distinguish the uses of that particular file number. The file sequence number is essential. It prevents a user from accidentally using the file ID of an already deleted file to access a file that was later given the same file number. File sequence numbers are assigned by maintaining the current sequence number in each file header and incrementing it each time the header is reused. If the previous value of the header's sequence number cannot be obtained, the file sequence number is generated randomly.
FID$B_RVN	Relative volume number. It indicates the volume of a volume set on which a file is located. If this volume is not part of a volume set, then this word contains a value of 0. If the volume is part of a volume set, then the relative volume number (RVN) can range from 1 to 255. When a file must be referenced in the context of the volume on which it lies, a relative volume number of 0 is used, regardless of the RVN that may be assigned to that volume.
FID$B_NMX	File number extension. This byte is the high-order part of the file number. Together with the FID$W_NUM field, it forms the complete 24-bit file number.

2.3.3 File Header

In addition to the file ID, every file on a Files–11 volume is described by a **file header**. It is not actually part of the file; rather, it is contained in the volume's index file (see Section 2.5.1). The file header is essentially a catalog of information about the file, such as where the data is located and how it is structured. It is used by the file system itself, RMS, the Dump Utility, and the Backup Utility.

The file header, also called the **header block**, contains all the information necessary to access the file, including the list of extents that make up the file. File extents describe where the file is physically located on the volume. If a file has a large number of extents, multiple file headers are used to describe them. The file header also contains such information as the file ownership, protection, creation date, and creation time.

The file header has six areas:

- Header area
- Ident area
- Map area
- Access control list area
- Reserved area
- End checksum

The first five areas vary in size, and only the header area and the end checksum are mandatory. Because the areas are variable in length, any software that processes these structures must check their length before accessing any fields. Fields contained within the fixed portion of the header (that is, the header area up to, but not including, the FH2$L_HIGHWATER field) can be assumed to be present.

The fixed portion includes any fields in the header area before the start of the ident area. The FH2$B_IDOFFSET field points to the ident area, which is the first available area in which to store data. In other words, the header may contain variable-length areas, but its size is fixed at 512 bytes.

A valid file header is defined by the following rules:

- The header end checksum in the FH2$W_CHECKSUM field must be correct. The end checksum (or block checksum) is a word occupying the last two bytes of the file header, and it is a simple additive checksum of all other words in the header block. It is verified every time a header is read and is recalculated every time a header is written.
- The value (that is, the address) contained in the FH2$B_IDOFFSET field is no less than the value represented by $\frac{FH2\$L\_HIGHWATER}{2}$.
- The four offset bytes are related such that the value contained in the FH2$B_IDOFFSET field is less than or equal to the value in the FH2$B_MPOFFSET field, which is less than or equal to the value in the FH2$B_ACOFFSET field, which is less than or equal to the value in the FH2$B_RSOFFSET field.
- The high byte of the FH2$W_STRUCLEV field contains the value 2.

- The low byte of the FH2$W_STRUCLEV field contains a value greater than or equal to 1.
- The word FH2$W_FID_NUM contains the file number.
- The word FH2$W_FID_SEQ contains the file sequence number.
- The high byte of the FH2$W_FID_RVN field (the FH2$B_FIX_NMX field) may contain the file number extension.
- The contents of the byte FH2$B_MAP_INUSE must be less than or equal to the value given by *FH2$B_ACOFFSET – FH2$B_MPOFFSET*.

A deleted file header conforms to the format of a valid file header, with the following exceptions:

- The FH2$V_MARKDEL bit is set in the FH2$L_FILECHAR field.
- The FH2$W_FID_NUM, FH2$B_FID_NMX, and the FH2$B_FID_RVN fields all contain a value of 0.
- The FH2$W_CHECKSUM field contains a value of 0.

2.3.3.1 Header Area

The header area of the file header always starts at byte 0. It contains information that allows the file system to make sure that this block is a valid file header and that this header is the correct one. It contains the file number and file sequence number of the file as well as its ownership and protection codes.

This area also contains offsets to the other areas of the file header, so it defines their size as well. Unlike the header area and the end checksum, the ident, map, access control list, and reserved areas are optional. If an area is not defined, the offset does not contain a value of 0; rather, the two offsets from which the size of the area can be calculated are equal. All areas except the end checksum are variable in length.

The symbol FH2$C_LENGTH contains the size of the header area, excluding the last field, FH2$R_CLASS_PROT.

The fields in the header area are illustrated in Figure 2–2, and each field is described in Table 2–2.

Figure 2–2: Format of the Header Area

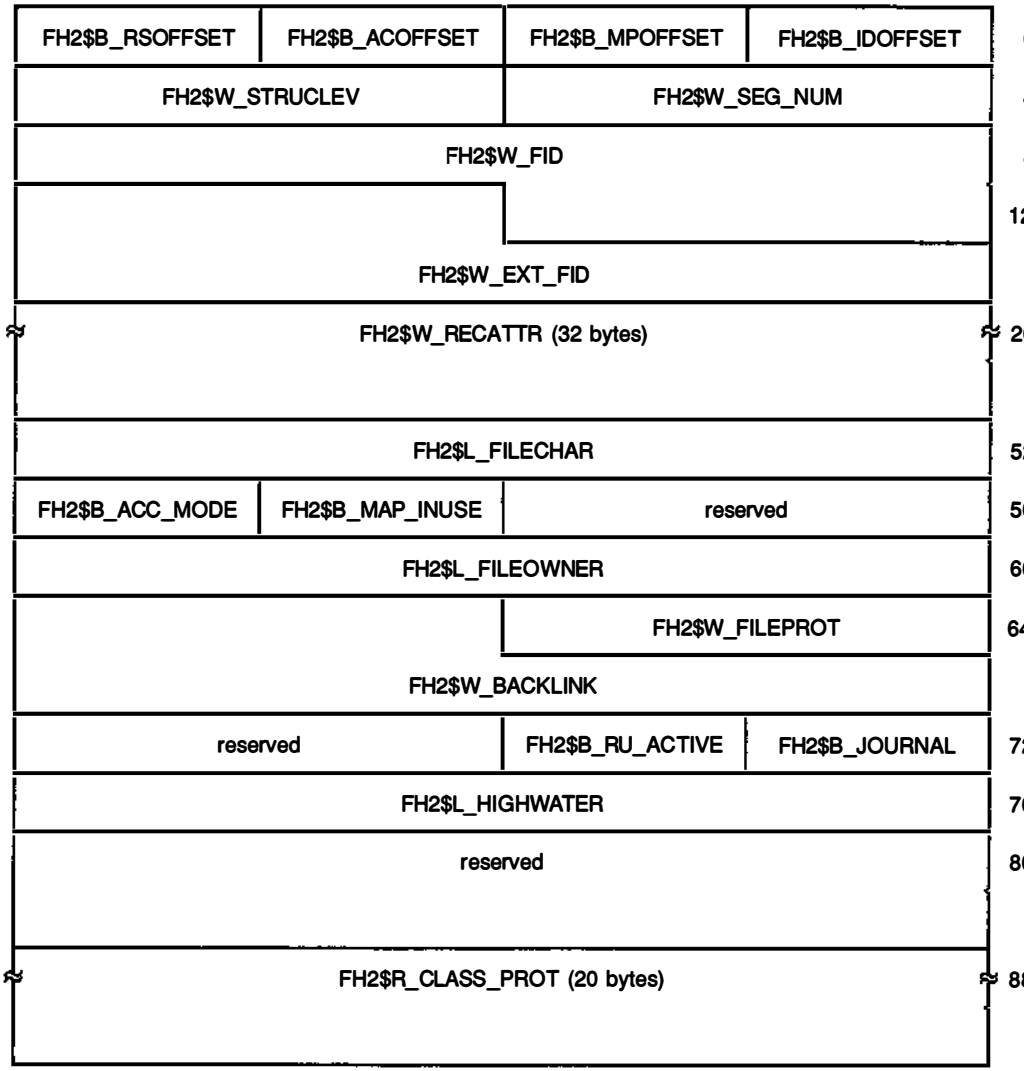


Table 2–2: Contents of the Header Area

Field Name	Description
FH2$B_IDOFFSET	Ident area offset. This byte contains the number of 16-bit words between the start of the file header and the start of the ident area. It defines both the location of the ident area and the size of the header area.
FH2$B_MPOFFSET	Map area offset. This byte contains the number of 16-bit words between the start of the file header and the start of the map area. It defines both the location of the map area and, with the FH2$B_IDOFFSET field, the size of the ident area.[1]
FH2$B_ACOFFSET	Access control list offset. This byte contains the number of 16-bit words between the start of the file header and the start of the access control list. It defines both the location of the access control list and, with the FD2$B_MPOFFSET field, the size of the map area.
FH2$B_RSOFFSET	Reserved area offset. This byte contains the number of 16-bit words between the start of the header and the start of the reserved area. The reserved area is not used by Files–11, so it may be used for special applications. With the FH2$B_ACOFFSET field, this byte defines the size of the access control list. The size of the reserved area can be calculated from the value in the FH2$B_RSOFFSET field and the end of the header block (excluding the end checksum).
FH2$W_SEG_NUM	Extension segment number. This word contains a value n, which indicates the header's ordinal position in the file. However, file headers are numbered sequentially starting with 0, so the header of value n is actually the header of position $n+1$ in the file. For example, a header defined by the value 2 is the third header in the file.

[1]Any free space in the file header should be allocated to the map area. In the primary header of a file, at least 8 bytes must be available for the map area.

(continued on next page)

Table 2–2 (Cont.): Contents of the Header Area

Field Name	Description
FH2$W_STRUCLEV	Structure level and version. This word is used to identify different versions of Files–11 because they affect the structure of the file header. This field also identifies the version of Files–11 used to create a file, which permits upward compatibility of file structures. Under the Files–11 On-Disk Structure Level 2, the high byte of this field must contain the value 2. The low byte contains the version number (currently version 1 of structure level 2), which must be greater than or equal to 1. The version number will be incremented when compatible additions are made to the Files–11 structure.
FH2$W_FID	File identifier. This field contains the file ID of the file. The format of a file ID is described in Section 2.3.2. This field contains the following four subfields: FH2$W_FID_NUM — Low-order file number FH2$W_FID_SEQ — File sequence number FH2$B_FID_RVN — Relative volume number. Because the file ID refers to itself (and therefore always points to the same volume), the value of this field is always 0. FH2$B_FID_NMX — High-order file number
FH2$W_EXT_FID	Extension file identifier. This field contains the file ID of the file's next extension header, if one exists. A value of 0 indicates that no extension header exists. This field contains the following four subfields: FD2$W_EX_FIDNUM — Low-order file number FH2$W_EX_FIDSEQ — File sequence number FH2$B_EX_FIDRVN — Relative volume number FH2$B_EX_FIDNMX — High-order file number
FH2$W_RECATTR	File record attributes. This area is used by the record manager or any other high-level access mechanism to store information necessary for processing the file, such as record control data or an end-of-file (EOF) mark.

(continued on next page)

Table 2–2 (Cont.): Contents of the Header Area

Field Name	Description	
FH2$L_FILECHAR	File characteristics. The following flag bits are defined relative to the start of this field:	
	FCH$V_NOBACKUP	Set if the file contents are not to be copied by the Backup Utility (BACKUP).
	FCH$V_WRITEBACK	Set if a write-back cache may be used. With this type of caching operation, the cached data is written to the disk only when it is removed from the cache. This bit is clear for write-through cache operations.
	FCH$V_READCHECK	Set if read-check operations are to be performed. All read operations on the file, including the file header, are verified with a read-compare operation to ensure data integrity.
	FCH$V_WRITCHECK	Set if write-check operations are to be performed. All write operations on the file, including modifications of the file header, are performed with a read-compare operation to ensure data integrity.
	FCH$V_CONTIGB	Set if the file is to be allocated contiguously in as few contiguous sections as possible. The storage bitmap is scanned for this purpose, causing the file system to perform extra I/O operations. The file system allocates the three largest contiguous pieces. If the request has not been satisfied, the file system disregards this bit and satisfies the request as best it can. The resulting allocation cannot be determined. This bit may be implicitly set or cleared by file system operations that allocate space to the file.

(continued on next page)

Table 2–2 (Cont.): Contents of the Header Area

Field Name	Description	
	FCH$V_LOCKED	Set if the file was locked on deaccess. This bit warns that the file was not properly closed and may contain inconsistent data. Access to the file is denied if this bit is set.
	FCH$V_CONTIG	Set if the file is logically contiguous (that is if, for all virtual blocks in the file, virtual block i maps to logical block $k + i$ on one volume for some constant k). This bit may be implicitly set or cleared by file system operations that allocate space to the file. The user may only clear it explicitly.
	FCH$V_BADACL	Set if the access control list of this file is not valid (if, for example, a system failure occurred while the list was being updated). In this case, the access control list for the file is not used for protection checking.
	FCH$V_SPOOL	Set if the file is a spool file (for example, a temporary storage area for files that are to be printed later). File operations not related to spool file handling are not allowed.
	FCH$V_DIRECTORY	Set if the file is a directory.
	FCH$V_BADBLOCK	Set if there is a bad data block in the file. It indicates that deferred bad block processing is to be done on the file at some suitable later time, such as after the file is deleted.
	FCH$V_MARKDEL	Set if the file is marked for deletion. If this bit is set, further access to the file is denied, and the file is physically removed from the disk after the last user has closed it.

(continued on next page)

Table 2–2 (Cont.): Contents of the Header Area

Field Name	Description	
	FCH$V_NOCHARGE	Set if the space used by this file is not to be charged to its owner.
	FCH$V_ERASE	Set if the file is to be erased or overwritten when it is deleted.
FH2$B_MAP_INUSE	Map words in use. This byte contains a count of the number of map area words currently in use.	
FH2$B_ACC_MODE	Accessor privilege level. This byte defines the lowest privilege level that an accessor must have to access the file.	
	Each privilege level is a 2-bit integer. A value of 0 indicates the lowest privilege and 3 the highest. Privilege levels may be assigned separately to the basic file access modes, using this bit assignment in the access mode byte:	
	Read	Bits <0:1>
	Write	Bits <2:3>
	Execute	Bits <4:5>
	Delete	Bits <6:7>
FH2$L_FILEOWNER	File owner identication. This field may be a user identification code (UIC) identifier or a general identifier. The file owner is usually, but not necessarily, the creator of the file.	
FH2$W_FILEPROT	File protection code. This word controls what access all users in the system can have to the file. When a user tries to open a file, the user's UIC is compared to the UIC of the owner of the file. Depending on the relationship of the UICs, the user may be classified in one or more of the categories below. Each category is controlled by a 4-bit field in the protection word.	

(continued on next page)

Table 2–2 (Cont.): Contents of the Header Area

Field Name	Description		
	System	Bits <0:3>	One of these four conditions must be met: • The group number of the user's UIC is less than or equal to the value set with the system parameter MAXSYSGRP (the default is 10_8). • The user holds SYSPRV privilege. • The user holds GRPPRV privilege and is in the same group as the file's owner. • The user is the owner of the volume.
	Owner	Bits <4:7>	The UIC exactly matches the file owner UIC.
	Group	Bits <8:11>	The group number of the UIC matches the group number of the file owner UIC. If a file is owned by a general identifier, however, group protection checking is not done.
	World	Bits <12:15>	The user does not fit into any of the categories above.

Four types of access are defined in Files–11: read (R), write (W), execute (E), and delete (D). Each 4-bit field in the protection word is bit-encoded to permit or deny any combination of the four types of access to that category of accessors. Setting a bit denies that type of access to that category. The bits within each 4-bit field have the following uses:

Bit <0> Set to deny read access.

Bit <1> Set to deny write access.

Bit <2> Set to deny execute access.

Bit <3> Set to deny delete access.

(continued on next page)

Table 2–2 (Cont.): Contents of the Header Area

Field Name	Description
	When a user tries to access a file, protection checks are performed in each category in this order: system, owner, group, and world. Access to the file is granted if any of the categories match. A fifth type of access—**control access**—governs the right to change the attributes of a file, such as its protection or its characteristics. In other words, control access allows modifications to the file header. Control access is not granted by a protection mask. It is always available to the system and owner categories but never to the group and world categories.
FH2$W_BACKLINK	Back link file ID. This field contains the file's back link pointer. It contains the file ID of the directory file that contains the primary directory entry for the file. If the file header is an extension header, the back link contains the file ID of the primary header. A value of 0 indicates that no back link exists. This field contains the following four subfields: FH2$W_BK_FIDNUM Low-order file number FH2$W_BK_FIDSEQ File sequence number FH2$W_BK_FIDRVN Relative volume number FH2$W_BK_FIDNMX High-order file number
FH2$B_JOURNAL	Journal control flags. This field is reserved. This field contains flags used to control the journaling facility provided by a high-level access method. The following flag bits are defined relative to the start of this field: FJN$V_ONLY_RU Set if the file is to be accessed only in a recovery unit. FJN$V_RUJNL Set if recovery-unit journaling is to be enabled. FJN$V_BIJNL Set if before-image journaling is to be enabled. FJN$V_AIJNL Set if after-image journaling is to be enabled. FJN$V_ATJNL Set if audit-trail journaling is to be enabled.

(continued on next page)

Table 2–2 (Cont.): Contents of the Header Area

Field Name	Description
	FJN$V_NEVER_RU — Set if the file is not to be accessed from within a recovery unit.
	FJN$V_JOURNAL_FILE — Set if the file is an RMS journal file.
FH2$B_RU_ACTIVE	Recoverable facility ID number. This field contains an identifier of the facility managing the file in an active recovery unit.
FH2$L_HIGHWATER	File highwater mark. This field contains the virtual block number, plus 1, of the highest block written. This form of security protection prevents blocks past this point from being read. If the highwater mark field contains 0, or if the header area is too short to contain this field (if the disk predates VMS Version 4.0), then no highwater mark has been maintained for the file. In this case, the file system does not use highwater marking.
FH2$R_CLASS_PROT	Security classification mask. This field contains the security classification of the file, which is used when the file system enforces a lattice-model security system incorporating the Bell and La Padula secrecy model and the Biba integrity model. This field is not currently supported. The classification mask block has the structure shown in Figure 2–3. The following field names are defined relative to the start of the classification mask block: CLS$B_SECUR_LEV — Secrecy level. This byte contains the secrecy classification level. A value of 0 indicates the least sensitive level, and 255 the most sensitive. CLS$B_INTEG_LEV — Integrity level. This byte contains the integrity classification level. A value of 0 indicates the least trustworthy level, and 255 the most trustworthy. CLS$Q_SECUR_CAT — Secrecy category mask. This 8-byte field contains a bit mask of the secrecy classes applicable to the file. In other words, to protect the confidentiality of the file, a user must hold the identifiers corresponding to the individual bits in this mask.

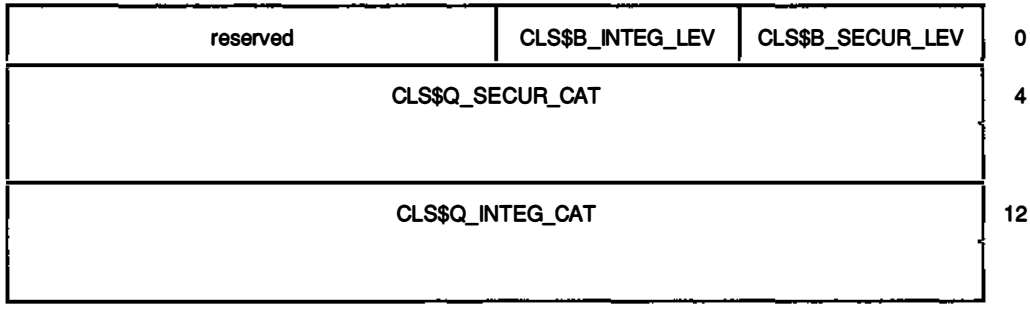
(continued on next page)

Table 2–2 (Cont.): Contents of the Header Area

Field Name	Description	
	CLS$Q_INTEG_CAT	Integrity category mask. This 8-byte field contains a bit mask of the integrity classes applicable to the file. In other words, to protect the veracity of the file, a user must hold the identifiers corresponding to the individual bits in this mask.

Figure 2–3 shows the classification mask block.

Figure 2–3: Format of the Classification Mask Block



2.3.3.2 Ident Area

The ident area of a file header is an optional area that stores the identification and accounting data about the file. It contains the primary name of the file; its creation date and time; revision count, date and time; expiration date and time; and backup date and time.

The ident area of the file begins at the word indicated by the FH2$B_IDOFFSET field. To allow more room for map pointers and access control list entries, the ident area is usually truncated in extension headers.

The symbol FI2$C_LENGTH contains the size of the ident area.

The field names of the ident area are illustrated in Figure 2–4 and are described in Table 2–3.

Figure 2–4: Format of the Ident Area

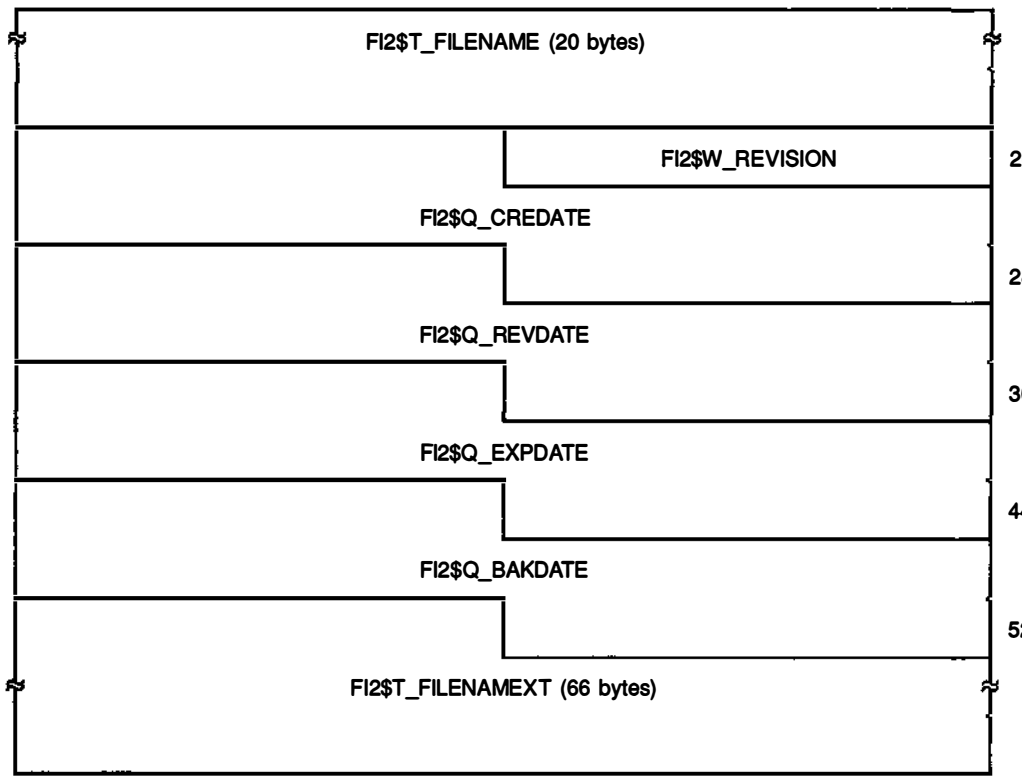


Table 2–3: Contents of the Ident Area

Field Name	Description
FI2$T_FILENAME	File name. This field contains the file name in ASCII form. A period separates the file name from the file type, and a semicolon separates the file type from the version number; both are always present. Names shorter than 20 bytes are padded with blanks. Longer names are continued in the FI2$T_FILENAMEXT field.
FI2$W_REVISION	Revision number. This word contains the revision count of the file in binary form. The revision count is the number of times the file has been accessed for writing.
FI2$Q_CREDATE	Creation date and time. These eight bytes contain the date and time at which the file was created. The time is expressed in the standard internal time format, which is a 64-bit integer representing tenths of microseconds elapsed since midnight, November 17, 1858.[1]
FI2$Q_REVDATE	Revision date and time. The revision time is the time at which the file was last closed after being accessed for write. It is expressed in the same format as the FI2$Q_CREDATE field.
FI2$Q_EXPDATE	Expiration date and time. These eight bytes contain the date and time at which the file becomes eligible for deletion. The format is the same as that of the FI2$Q_CREDATE and FI2$Q_REVDATE fields above.
FI2$Q_BAKDATE	Backup date and time. These eight bytes contain the date and time at which the file was last backed up. The format is the same as the other dates and times.
FI2$T_FILENAMEXT	File name extension. This field contains the remainder of the file name, continued from the FI2$T_FILENAME field above. The 86-character file name field allows for the 80 characters of the file name and type, plus the five digits of the version number.

[1]This date, base date for the Smithsonian astronomical calendar, is derived from the Julian Date. Julian Date (JD) is used by astronomers and is expressed in days elapsed since January 1, 4713 B.C. The modified Julian Date base of JD 2,400,000 was adopted by the Smithsonian Astrophysical Observatory for satellite tracking, and this date corresponds to the Julian date of November 17, 1858.

2.3.3.3 Map Area

The map area of the file header is an optional area that contains the information necessary to map the virtual blocks of the file to the logical blocks of the volume. This area of the file header starts at the word indicated by the FH2$B_MPOFFSET field.

The map area consists of a list of **retrieval pointers** or **map pointers** describing the logical blocks allocated to the file. Retrieval pointers are listed in the order of the virtual blocks they represent.

Each retrieval pointer describes an **extent**, a consecutively numbered group of logical blocks allocated to the file. The count field of the pointer contains the binary value n, which represents a group of $n + 1$ logical blocks. The logical block number field contains the logical block number of the first logical block in the group.

Mapping can be described with the following formula:

- j is the total number, plus 1, of the virtual blocks represented by all preceding retrieval pointers in the current and all preceding headers of the file.
- k is the value contained in the logical block number field.
- n is the value contained in the count field.

Given the arguments in the previous list, then each retrieval pointer maps virtual blocks j through $j + n$ into logical blocks k through $k + n$, respectively. Note, however, that j, k, and $n + 1$ must always be integer multiples of the volume cluster factor (the minimum disk allocation unit in blocks). The cluster factor default is 3 for disks larger than 50,000 blocks. Otherwise, the default is 1.

Retrieval pointers have four formats, and they may be intermixed within a file header. The format code of each retrieval pointer is contained in the FM2$V_FORMAT field, which represents the two high-order bits of the first word.

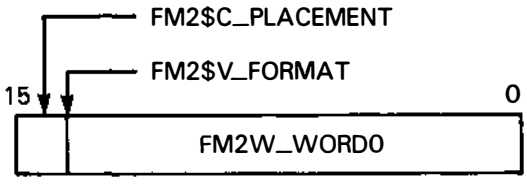
The four formats are described in the following sections.

2.3.3.3.1 Retrieval Pointer Format 0

Retrieval pointer format 0 describes a structure called a **placement header**. This 2-byte field is represented by the value FM2$C_PLACEMENT, which means the FM2$V_FORMAT bit contains a value of 0(binary). It stores information in the file header about a file's allocation. It records the placement options selected when the file was created so that the conditions of the allocation may be duplicated.

The format of this retrieval pointer is illustrated in Figure 2–5.

Figure 2–5: Retrieval Pointer Format 0



A placement header, denoted by the FM2$W_WORD0 field, contains the following placement control bits.

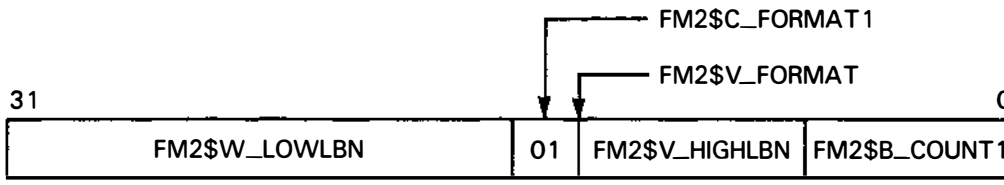
Bit Name	Description
FM2$V_EXACT	Set if exact placement is requested or if space must be allocated as specified.
FM2$V_ONCYL	Set if space is to be allocated on one cylinder of the volume.
FM2$V_LBN	Set if space is to be allocated at the start of the LBN contained in the next retrieval pointer in the list.
FM2$V_RVN	Set if space is to be allocated on the specified volume (the volume on which this extent is located).

2.3.3.3.2 Retrieval Pointer Format 1

Retrieval pointer format 1 is represented by the value FM2$C_FORMAT1, which means that FM2$V_FORMAT contains a value of 1(binary). This 4-byte field provides an 8-bit count field and a 22-bit LBN field. It is therefore capable of representing a group of up to 256 blocks on a volume of up to 2^{22} blocks in size.

The format of this retrieval pointer is illustrated in Figure 2–6.

Figure 2–6: Retrieval Pointer Format 1



The following three field names are associated with this format.

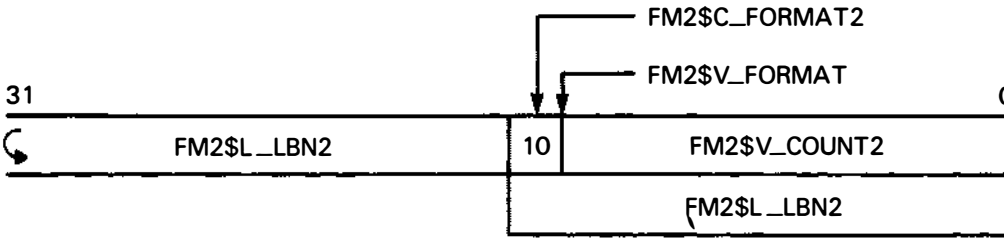
Field Name	Description
FM2$B_COUNT1	Block count. This 8-bit count field contains the binary value *n*, which represents a group of *n* + 1 logical blocks.
FM2$W_LOWLBN	Low-order LBN. This field contains the logical block number of the first LBN in the group described by the count field. This field and the FM2$V_HIGHLBN field form the total 22-bit LBN.
FM2$V_HIGHLBN	High-order LBN. This field contains the high 6 bits of the logical block number. This field and the FM2$W_LOWLBN field form the total 22-bit LBN.

2.3.3.3.3 Retrieval Pointer Format 2

Retrieval pointer format 2 is represented by the value FM2$C_FORMAT2, which means that FM2$V_FORMAT contains a value of 2(binary). This 6-byte field provides a 14-bit count field and a 32-bit LBN field. It is capable of representing a group of up to 16,384 blocks on a volume of up to 2^{32} blocks.

The format of this retrieval pointer is illustrated in Figure 2–7.

Figure 2–7: Retrieval Pointer Format 2



The following two field names are associated with this format.

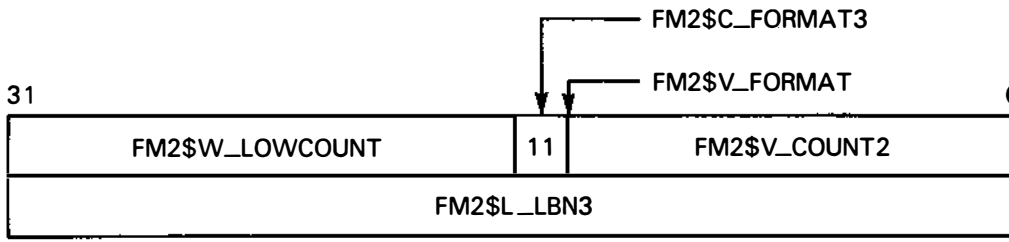
Field Name	Description
FM2$V_COUNT2	Block count. This 14-bit count field contains the binary value *n*, which represents a group of *n* + 1 logical blocks.
FM2$L_LBN2	LBN. This 32-bit field contains the logical block number of the first LBN in the group described.

2.3.3.3.4 Retrieval Pointer Format 3

Retrieval pointer format 3 is represented by the value FM2$C_FORMAT3, which means that FM2$V_FORMAT contains a value of 3(binary). This 8-byte field provides a 30-bit count field and a 32-bit LBN field. It is capable of describing a group of up to 2^{30} blocks on a volume of up to 2^{32} blocks.

The format of this retrieval pointer is illustrated in Figure 2–8.

Figure 2–8: Retrieval Pointer Format 3



ZK-9583-HC

The following three field names are associated with this format.

Field Name	Description
FM2$V_COUNT2	High-order 14 bits of the count field. This 30-bit count field contains the binary value n, which represents a group of $n + 1$ logical blocks.
FM2$W_LOWCOUNT	Low-order 16 bits of the count field. FM2$W_LOWCOUNT and FM2$V_COUNT2 form the total 30-bit count field.
FM2$L_LBN3	Logical block number. This 32-bit field contains the logical block number of the first LBN in the group described by the count field.

2.3.3.4 Access Control List Area

The access control list (ACL) area is an optional area containing, among other things, a list of users or identifiers who are allowed to access a file. The access control list can describe user communities for a particular file that cannot be expresssed with the regular protection classes.

This area may also be used for storing additional information about the file.

The individual access control entries (ACEs) are stored in the ACL area with no surrounding structure. The access control list may span multiple headers, occupying the ACL area in each header.

An ACE cannot cross a file header. A single ACE is limited to a total of 256 bytes (including the size, type, flags, and access fields).

There may be up to 255 types of ACEs, but only five types are currently supported. All other types are reserved. The five supported types are as follows:

- Alarm ACE—See Section 2.3.3.4.1
- Application ACE—See Section 2.3.3.4.2

- Directory default protection ACE—See Section 2.3.3.4.3
- Identifier ACE—See Section 2.3.3.4.4
- RMS journaling ACE—See Section 2.3.3.4.5

Each has a different format, and each is referenced by a unique symbolic constant.

However, every ACE structure contains the basic fields shown in Figure 2–9 and described in Table 2–4. Note, however, that access bits are stored true, as opposed to existing protection masks. In other words, a bit containing a value of 0 (clear) denies access, while a value of 1 (set) grants access.

Figure 2–9: Format of the Basic Access Control Entry

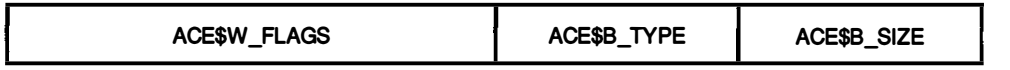


Table 2–4: Contents of the Basic Access Control Entry

Field Name	Description
ACE$B_SIZE	ACE size. This size includes the overhead area and all of the keys.
ACE$B_TYPE	ACE type. This type code determines how the remainder of the ACE is interpreted. A related field, ACE$W_FLAGS, contains both type-dependent and type-independent flags. The type codes are as follows:
	ACE$C_ALARM — The name of the journal to which a security alarm is to be written when the file is accessed successfully or unsuccessfully. (See ACE$V_SUCCESS and ACE$V_FAILURE for more information.)

(continued on next page)

Table 2–4 (Cont.): Contents of the Basic Access Control Entry

Field Name	Description	
	ACE$C_AUDIT	The name of the journal to which a security audit journal record is to be written when the file is either successfully or unsucessfully accessed. This format is not supported by VMS Version 5.0.
	ACE$C_DIRDEF	The ACE contains default protection for files created in a directory. This protection information is used instead of the process default protection, unless it is explicitly overridden.
	ACE$C_INFO	The ACE contains general or application-dependent information. The maximum length of the information that can be stored is 252 bytes although there is no limit to the number of INFO ACEs that may appear in a file's ACL. (See ACE$V_INFO_TYPE for more information.)
	ACE$C_KEYID	The longword identifiers used to determine who may gain access to the file. One type of identifier is the UIC identifier. The other types of identifiers are general and system-defined. The maximum number of identifiers that can be defined is 62. (See ACE$V_RESERVED for more information.)
	ACE$C_RMSJNL_AI	The location of the RMS after-image journal.

(continued on next page)

Table 2–4 (Cont.): Contents of the Basic Access Control Entry

Field Name	Description
	ACE$C_RMSJNL_AT — The location of the RMS audit-trail journal.
	ACE$C_RMSJNL_BI — The location of the RMS before-image journal.
	ACE$C_RMSJNL_RU — The location of the RMS recovery-unit journal.
	ACE$C_RMSJNL_RU_DEFAULT — The location of the default RMS recovery-unit journal.
ACE$W_FLAGS	Type flags. This field, along with the ACE$B_TYPE field, determines how the ACE is used. This word is divided into two 1-byte fields: type-dependent flags and type-independent flags. The type-dependent flags augment the type code, which allows many different subtypes to be defined. The type-independent flags are used to provide features that do not depend on the ACE type. The type-dependent flags are as follows: ACE$V_INFO_TYPE — This 4-bit flag contains a value to indicate a subtype of the general information ACE. The following three values are defined for this field: • The ACE$C_CUST value indicates that the information belongs to a user application. • The ACE$C_CSS value indicates that the information belongs to a Digital Computer Special Services (CSS) application. • The ACE$C_VMS value indicates that the information is valid only for a specific VAX/VMS utility, application, or layered product.

(continued on next page)

Table 2–4 (Cont.): Contents of the Basic Access Control Entry

Field Name	Description	
	ACE$V_RESERVED	This flag is valid only for the ACE$C_KEYID type. See Section 2.3.3.4.4 for more information.
	ACE$V_SUCCESS	This flag is valid only for the ACE$C_ALARM type. See Section 2.3.3.4.1 for more information.
	ACE$V_FAILURE	This flag is valid only for the ACE$C_ALARM type. See Section 2.3.3.4.1 for more information.
	The type-independent flag definitions are as follows:	
	ACE$V_DEFAULT	The ACE is a directory default ACE. This ACE is applied to all files created in a common directory. When the ACE is propagated, the DEFAULT option is removed from the ACE before it is added to the ACL of the created file. This option is valid only for directory files. A default ACE does not control access to the directory to which it belongs. It only specifies the ACL for files created in that directory.
	ACE$V_PROTECTED	The ACE will not be deleted when the ACL for the file is deleted (by the ATR$C_DELETEACL attribute code). Instead, the ACE must be deleted explicitly with the ATR$C_DELACLENT attribute code.
	ACE$V_HIDDEN	The ACE is not one that the user should usually see, but it may be used by some application. The DIRECTORY command, for example, does not display it.
	ACE$V_NOPROPAGATE	This ACE should not be copied during any form of ACL copy operation, either from one version of a file to a later version of the same file or from the parent directory to a newly created file within it.

2.3.3.4.1 Alarm Access Control Entry

The alarm ACE provides a security alarm when an object is accesssed in a particular way. It may be referenced by the symbol ACE$C_ALARM. Figure 2–10 shows the format of the alarm ACE type, and Table 2–5 describes the fields unique to this format.

Figure 2–10: Format of the Alarm ACE

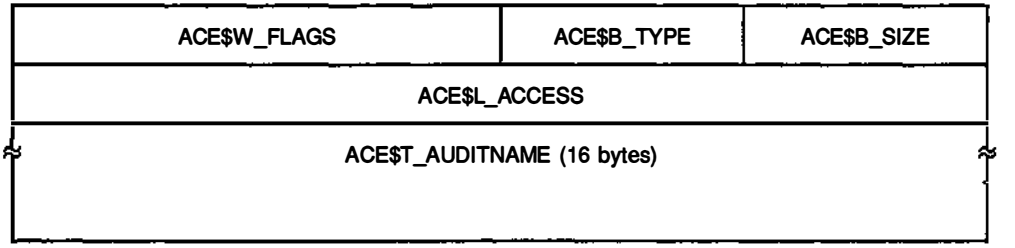


Table 2–5: Contents of the Alarm ACE

Field Name	Description	
ACE$B_SIZE	ACE size.	
ACE$B_TYPE	ACE type.	
ACE$W_FLAGS	Type flags. The following two type-dependent flags are valid for the alarm ACE:	
	ACE$V_SUCCESS	This field specifies that a security alarm should be generated when access is granted to a file. This flag is valid only for the ACE$C_ALARM type.
	ACE$V_FAILURE	This field specifies that a security alarm should be generated when access is denied to a file. This flag is valid only for the ACE$C_ALARM type.

(continued on next page)

Table 2–5 (Cont.): Contents of the Alarm ACE

Field Name	Description	
ACE$L_ACCESS	Access type. This longword specifies the type of access for which a security audit or alarm message is to be issued. The following access rights are defined:	
	ACE$V_READ	A message is issued if read access to the file is attempted.
	ACE$V_WRITE	A message is issued if write access to the file is attempted.
	ACE$V_EXECUTE	A message is issued if execute access to the file is attempted.
	ACE$V_DELETE	A message is issued if delete privilege for the file is attempted.
	ACE$V_CONTROL	A message is issued if any of the rights of the file's owner are attempted.
ACE$T_AUDITNAME	Alarm journal name. This field is the start of the alarm-journal-name counted string.	

2.3.3.4.2 Application Access Control Entry

The application ACE (also called the INFO ACE) contains application-dependent or user-defined information. It may be referenced by the symbol ACE$C_INFO. Figure 2–11 shows the format of the application ACE type, and Table 2–6 describes the fields unique to this format.

Figure 2–11: Format of the Application ACE

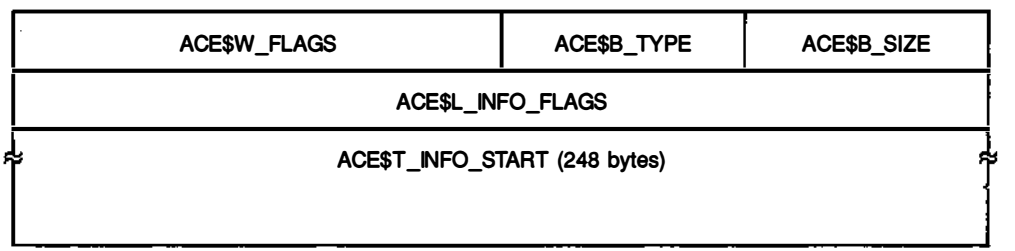


Table 2–6: Contents of the Application ACE

Field Name	Description
ACE$B_SIZE	ACE size.
ACE$B_TYPE	ACE type.
ACE$W_FLAGS	Type flags. No type-dependent flags are valid for the application ACE.
ACE$L_INFO_FLAGS	Application flags. This longword is used for VMS-specific application ACEs. This field is currently interpreted as two VMS-specific word subfields. RMS, for example, uses an application ACE of this type as an extension of the file header to store more information about a file, such as file statistics (see Figure 2–12 and Table 2–7 for more information). However, other applications are not restricted from using these application-defined subfields. The following word subfields are defined: ACE$W_APPLICATION_FLAGS — VMS application flags field. ACE$W_APPLICATION_FACILITY — VMS application facility field.
ACE$T_INFO_START	Information area. This location is the start of the application-dependent information area. The RMS attributes ACE uses this portion of the application ACE to store additional information about a file. Figure 2–12 shows the format of the RMS attributes ACE type, and Table 2–7 describes the fields unique to this format.

Figure 2–12: Format of the RMS Attributes ACE

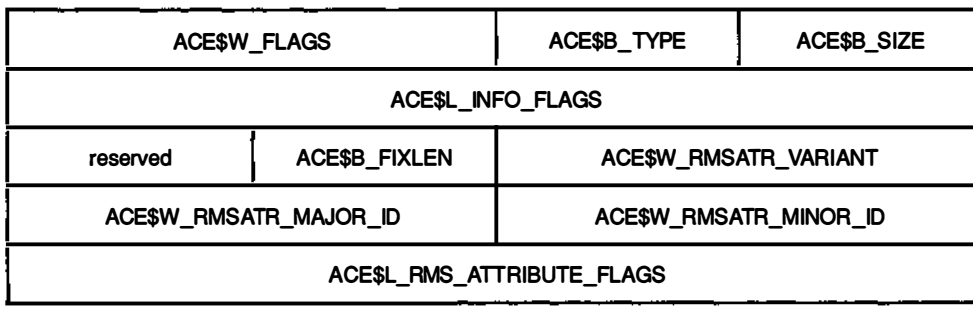


Table 2–7: Contents of the RMS Attributes ACE

Field Name	Description
ACE$B_SIZE	ACE size.
ACE$B_TYPE	ACE type.
ACE$W_FLAGS	Type flags. No type-dependent flags are valid for the application ACE.
ACE$L_INFO_FLAGS	Application flags.
ACE$W_RMSATR_VARIANT	Variant of the RMS attributes ACE. This field is currently set to 0.
ACE$B_RMSATR_FIXLEN	Fixed-format length. This field contains the length of the fixed portion of the ACE in bytes, which is currently 20 bytes.
ACE$W_RMSATR_MINOR_ID	RMS file attributes ACE minor identifier. This field contains an integer that identifies the current version of the ACE. This number is incremented when compatible changes to the ACE have been made. For VMS Version 5.0, this field is set to 2.
ACE$W_RMSATR_MAJOR_ID	RMS file attributes ACE major identifier. This field contains an integer that identifies the current version of the ACE. This number is incremented when incompatible changes to the ACE have been made. For VMS Version 5.0, this field is set to 1.
ACE$L_RMS_ATTRIBUTE_FLAGS	RMS file attributes flags definitions. The following flags are defined for this field: STATISTICS — If set, statistics monitoring is enabled for the file. XLATE_DEC — If set, file semantics are private to Digital. This field is not supported for VMS Version 5.0.

2.3.3.4.3 Directory Default Protection Access Control Entry

The directory default protection ACE (also called the DIRDEF ACE) defines the default protection for a directory so that protection can be propagated to the files and subdirectories in that directory. This type of ACE specifies protection for one directory structure that is different from the default protection applied to other directories. Default protection ACEs can be applied only to directory files.

This ACE may be referenced by the symbol ACE$C_DIRDEF. Figure 2–13 shows the format of the directory default protection ACE type, and Table 2–8 describes the fields unique to this format.

Figure 2–13: Format of a Directory Default Protection Ace

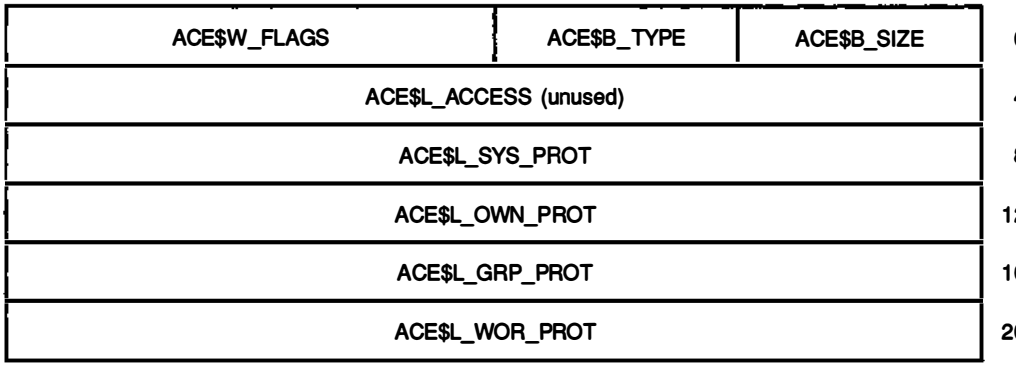


Table 2–8: Contents of the Directory Default Protection ACE

Field Name	Description
ACE$B_SIZE	ACE size.
ACE$B_TYPE	ACE type.
ACE$W_FLAGS	Type flags. No type-dependent flags are valid for the directory default protection ACE.
ACE$L_ACCESS	Access type. This longword is unused.
ACE$L_SYS_PROT	Directory default system protection.
ACE$L_OWN_PROT	Directory default owner protection.

(continued on next page)

Table 2–8 (Cont.): Contents of the Directory Default Protection ACE

Field Name	Description
ACE$L_GRP_PROT	Directory default group protection.
ACE$L_WOR_PROT	Directory default world protection.

2.3.3.4.4 Identifier Access Control Entry

The identifier ACE (also called the KEYID ACE) controls the type of access allowed to a particular user or group of users as specified by an **identifier**. Each ACE contains a list of identifiers that control access to the file or volume. An identifier is a logical extension to a UIC because it defines a particular entity in the system. The identifier may represent a UIC or a process rights list entry. In order to match a particular ACE, all of the identifiers must successfully match the corresponding identifiers of the accessor. If no ACE is matched or an ACL is not associated with the file, access is then granted or denied based on the conventional protection fields.

The identifier ACE is referenced by the symbol ACE$C_KEYID. Figure 2–14 shows the format of the identifier ACE type, and Table 2–9 describes the fields unique to this format.

Figure 2–14: Format of the Identifier ACE

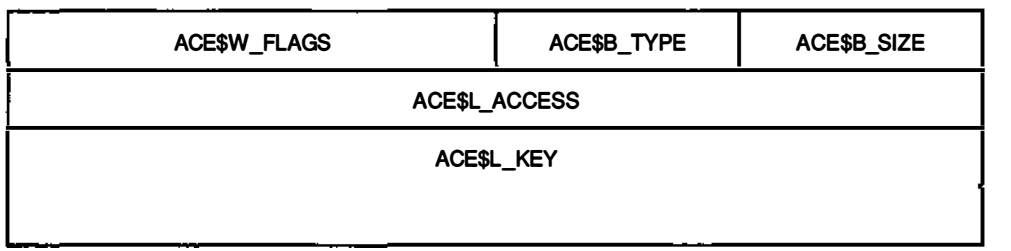


Table 2–9: Contents of the Identifier ACE

Field Name	Description
ACE$B_SIZE	ACE size.
ACE$B_TYPE	ACE type.
ACE$W_FLAGS	Type flags. One type-dependent flag is valid for the identifier ACE: ACE$V_RESERVED. This 4-bit flag field indicates the number of longwords to reserve for CSS or the user. Up to 15 longwords may be reserved. The reserved area starts at the ACE$L_KEY field. The actual keys then follow. This field is valid only for the ACE$C_KEYID type.
ACE$L_ACCESS	Access type. This longword specifies the type of access to be granted if all the keys are matched. The following access rights are defined: ACE$V_READ — Read access to the file is granted. ACE$V_WRITE — Write access to the file is granted. ACE$V_EXECUTE — Execute access to the file is granted. ACE$V_DELETE — Delete privileges for the file are granted. ACE$V_CONTROL — The right to change the protection and file characteristics of the file (that is, access to the file header) is granted.
ACE$L_KEY	Key field. This longword is the start of the variable-length key field. The number of identifiers listed in the ACE is implied by its size.

2.3.3.4.5 RMS Journaling Access Control Entries

RMS journaling provides a way to store changes that have been made to a file. Some journal information may be stored in access control entries. There are five types of RMS journaling access control entries. Table 2–10 shows each type of ACE, the symbolic name by which each may be referenced, and a description of each one.

Table 2–10: RMS Journaling ACE Types

Type of ACE	Description
RMSJNL_AI	The location of the after-image journal. A BI journal records changes to a file so that the journal can be used to roll the current copy of the file backward. In other words, the changes are undone. The journal name must be specified with the SET FILE/AI_JOURNAL command before a file is modified. Only one AI ACE may exist per file. The symbolic name is ACE$C_RMSJNL_AI.
RMSJNL_AT ACE	The location of the audit-trail journal. An AT journal records information about file and record accesses. Only one AT ACE may exist per file. The symbolic name is ACE$RMSJNL_AT.
RMSJNL_BI	The location of the before-image journal. An AI journal records changes to a file so that the journal can be used to roll a backup copy of the file forward. In other words, the changes are redone. The journal name must be specified with the SET FILE/BI_JOURNAL command before a file is modified. Only one BI ACE may exist per file. The symbolic name is ACE$C_RMSJNL_BI.
RMSJNL_RU	The location of a particular instance of a journal file. An RU journal records changes made during a recovery unit so that if the recovery unit aborts, the changes can be undone. The RU ACE is created at run time when a process uses the file in a recovery unit for the first time. It is removed when the file is closed. The RU ACE does not represent a particular journal stream; instead, it represents the existence of a potentially active transaction. Multiple RU ACES may exist per file. The symbolic name is ACE$C_RMSJNL_RU.
RMSJNL_RU_DEFAULT	The default location of a new journal file. The RU_DEFAULT ACE allows the RU journal to be created on a different volume than the volume on which the data file is being journaled, chiefly for performance reasons. If this ACE is not set, the RU journal is created on the same volume as the data file being journaled. Only one RU_DEFAULT ACE may exist per file. The symbolic name is ACE$C_RMSJNL_RU_DEFAULT.

All these ACEs are created as hidden, protected, and nopropagate.

These ACEs all have the same format, but not all fields apply to each type of ACE. For example, the RMSJNL_RU_DEFAULT ACE uses only the ACE$T_VOLNAM field. Figure 2–15 shows the format of these ACEs, and Table 2–11 describes the fields unique to this format.

Figure 2–15: Format of the RMS Journaling ACEs

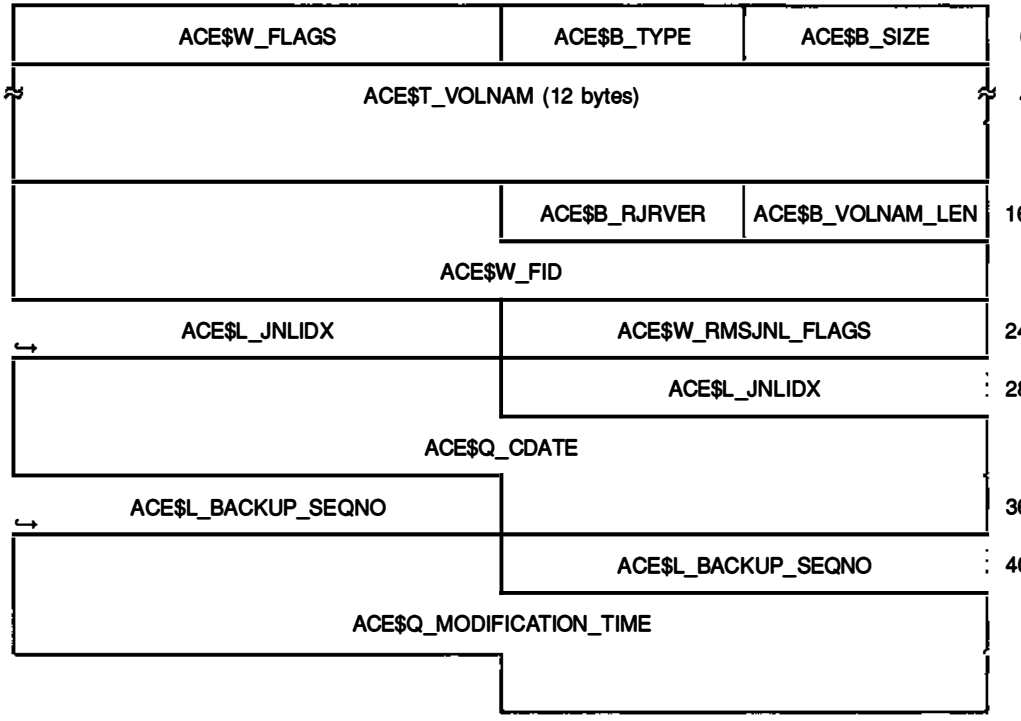


Table 2–11: Contents of the RMS Journaling ACEs

Field Name	Description
ACE$B_SIZE	ACE size.
ACE$B_TYPE	ACE type.

(continued on next page)

Table 2–11 (Cont.): Contents of the RMS Journaling ACEs

Field Name	Description
ACE$W_FLAGS	Type flags.
ACE$T_VOLNAM	Volume name. This 18-byte field contains the volume label in ASCII form, padded to 12 bytes with spaces, plus the 6-byte binary file ID. The RU_DEFAULT ACE uses only the basic ACE fields plus this field.
ACE$B_VOLNAM_LEN	Length of the journal file volume name in bytes. In other words, this field contains the length of the nonblank portion of the ACE$T_VOLNAM field.
ACE$B_RJRVER	RMS journal file structure level. This field contains the version number of the journal format. Currently, this field is set to 1.
ACE$W_FID	File identifier of the journal file. The format of a file ID is described in Section 2.3.2.
ACE$W_RMSJNL_FLAGS	RMS journaling flags. The following flags are defined: • ACE$V_JOURNAL_DISABLED—Indicates that journaling has been disabled. This bit applies only to after-image, before-image, and audit-trail journaling, and it is set by the Backup Utility. • ACE$V_BACKUP_DONE—Indicates that this file has been backed up and that RMS needs to write a backup marker to the journal file.
ACE$L_JNLIDX	Journal stream index number. More than one file can post entries to a journal file. This field contains a unique number used to distinguish the entries posted for one file from the entries posted for another.
ACE$Q_CDATE	Creation date and time of the journal file, in standard VMS format.

(continued on next page)

Table 2–11 (Cont.): Contents of the RMS Journaling ACEs

Field Name	Description
ACE$L_BACKUP_SEQNO	Backup sequence number. This field indicates where to start in the journal. In other words, this field is incremented each time RMS posts a backup marker to the journal file. This number is compared to the backup marker to determine if the journal entries bracketed by the backup markers need to be processed.
ACE$Q_MODIFICATION_TIME	Time-stamp of the last backup or last journal entry recovered, in standard VMS format.

2.3.3.5 User-Reserved Area

The optional reserved area of the file header starts at the word indicated by the byte FH2$B_RSOFFSET. It is only available in the primary header. This area is not used by the Files–11 file system, so it can be used by Computer Special Services (CSS) or a user's applications.

Application-dependent information may also be stored in an information ACE.

2.3.4 Multiheader Files

The size of the file header is fixed, so the mapping or access control information for some files will not fit in the allocated space. A file in which the information overflows the allocated space is called a **multiheader file**. It is represented by a chain of file headers called an **extension linkage**.

Each header in the chain maps a consecutive set of virtual blocks, and the extension linkage links the headers together in the order of ascending virtual block numbers. The extension pointer (located in the FH2$W_EXT_FID field) in each file header is the file ID of the next header in the sequence.

The access control list segments in the various headers are likewise considered one large access control list, concatenated in the order of the file headers. Access control lists and arrays of map pointers may be intermixed in the various headers of a file. In other words, each header may contain any amount of map or access control data regardless of its position in the sequence.

Technically, each header of a multiheader file could be accessed as a file because it has a file ID of its own. However, since the complete access control information is visible only from the primary header of a file, the file system prevents access to extension headers as files so that file protection can be correctly enforced.

2.3.5 Multivolume Files

Multiple headers are also needed for files that span volumes in a volume set. A header maps only those logical blocks of a file located on its volume. However, a **multivolume file** is represented by a header on each volume that contains a portion of that file.

If the multivolume file is contained on a loosely coupled volume set, the file ID of the first header on each continuation volume always has the value 7, 7, *n*, where *n* is the RVN of the volume on which the file starts plus the number of preceding volumes containing portions of the file.

2.4 Basic Concept of a Directory

A **directory** is simply a file used to locate other files on a volume. It contains a list of files and their unique internal identifications.

Files–11 provides directories to allow the organization of files in a meaningful way. The file ID uniquely locates a file on a volume set, but it is not very easy to remember. Directories are the special files that match alphabetic names with file identifiers.

2.4.1 Directory Structure

A directory file is always contiguous, and it is identified as a directory by the set FCH$V_DIRECTORY bit in the file characteristics field (FH2$L_FILECHAR). It is organized as a sequential file with variable-length records. The FAT$V_NOSPAN bit is also set, specifying that records may not cross block boundaries. No carriage control attributes are set.

The last word of the directory file's record attributes area (FAT$W_VERSIONS) is used to store the directory's default version limit. This version limit is assigned to all new files created in the directory if a version limit is not specified by the creator.

Directory records within each block of the directory file are packed together to conform to the variable-length record format. At the end of the sequence of records in each block, a word containing –1 signals the end of records for that block. This word is always present in a directory block, but it is optional in the variable-length record format itself.

The entries in a directory are sorted alphabetically, which allows for optimized searching. Entries which are multiple versions of the same name and type are arranged in order of decreasing version numbers to optimize version-related operations. Each directory record consists of the fields shown in Figure 2–16.

Figure 2–16: Format of a Directory Record

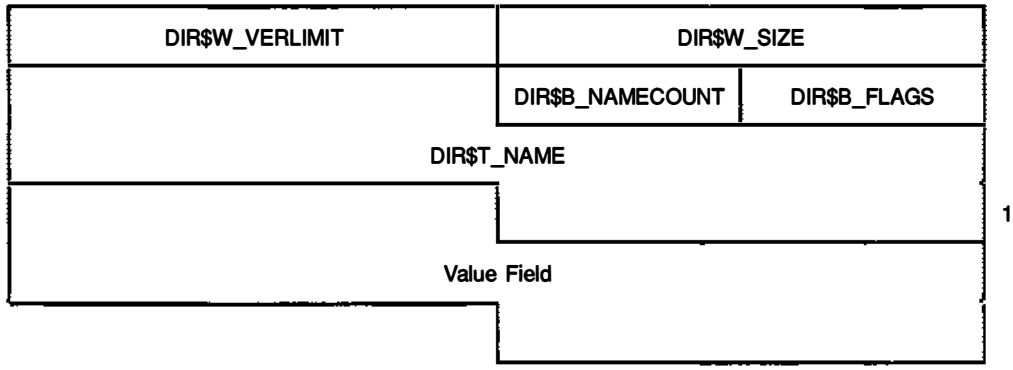


Table 2–12 shows the contents of a directory record.

Table 2–12: Contents of a Directory Record

Field Name	Description
DIR$W_SIZE	Record byte count. This field is the standard byte count field of a variable length record. It contains the length in bytes of the record (not including the two bytes of the count). The byte count is always an even number.
DIR$W_VERLIMIT	File version limit. This word contains the maximum number of versions that are retained for this file name and type. An attempt to create more versions than the limit will either cause the oldest version of the file to be deleted, or it will cause an error to be returned (if the oldest version cannot be deleted).
DIR$B_FLAGS	Flags. This byte contains the type code of the directory entry and assorted flag bits. It contains the following subfields and status bits: DIR$V_TYPE — The type code is contained in the three low bits of the flags byte. DIR$C_FID — The value field is a list of version numbers and 48-bit file identifiers.
DIR$B_NAMECOUNT	File name length. This field contains the length (in bytes) of the file name.

(continued on next page)

Table 2–12 (Cont.): Contents of a Directory Record

Field Name	Description
DIR$T_NAME	File name string. This field contains the file name and file type in ASCII form, separated by a period. The period is present even if either the name or the type, or both, are null. The file name and type may be composed of any of the standard name characters: alphanumerics (including the Multinational Character Set), the dollar sign (reserved for special use by Digital), the underscore, and the hyphen. The name and type fields are each limited to 39 characters. If the length of the name is an odd number, the name string is padded with a single null character.
Value field	Directory entry information. This variable field contains the information returned to the user by a directory lookup operation. Its interpretation depends on the directory record type. For a file ID record type (the type field is DIR$C_FID), a list of version numbers and corresponding file identifiers, in descending order by version number, is returned to the value field. The number of entries in the list can be calculated with this formula: $$recordlength - overhead - namestring = entries$$ Figure 2–17 and Table 2–13 describe the format and contents of an individual directory entry.

Figure 2–17: Format of a Directory Entry

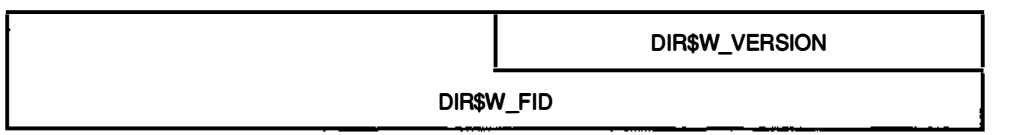


Table 2–13: Contents of a Directory Entry

Field Name	Description
DIR$W_VERSION	Version number. This word contains the version number of the directory entry in binary form. Version numbers can range from 1 to 32,767.
DIR$W_FID	File ID. These three words are the file identifier to which the directory entry points.

2.4.2 Multiple Directory Records

Directory records may not cross block boundaries, so there is a limit to the number of file versions that can be contained in one directory record. That number is 62 for the shortest possible file name.

To represent more versions of a file than will fit into one directory block, multiple directory records are used. The records are ordered by descending version numbers, as are the versions within each record. Each record contains the full file name.

2.4.3 Directory Hierarchies

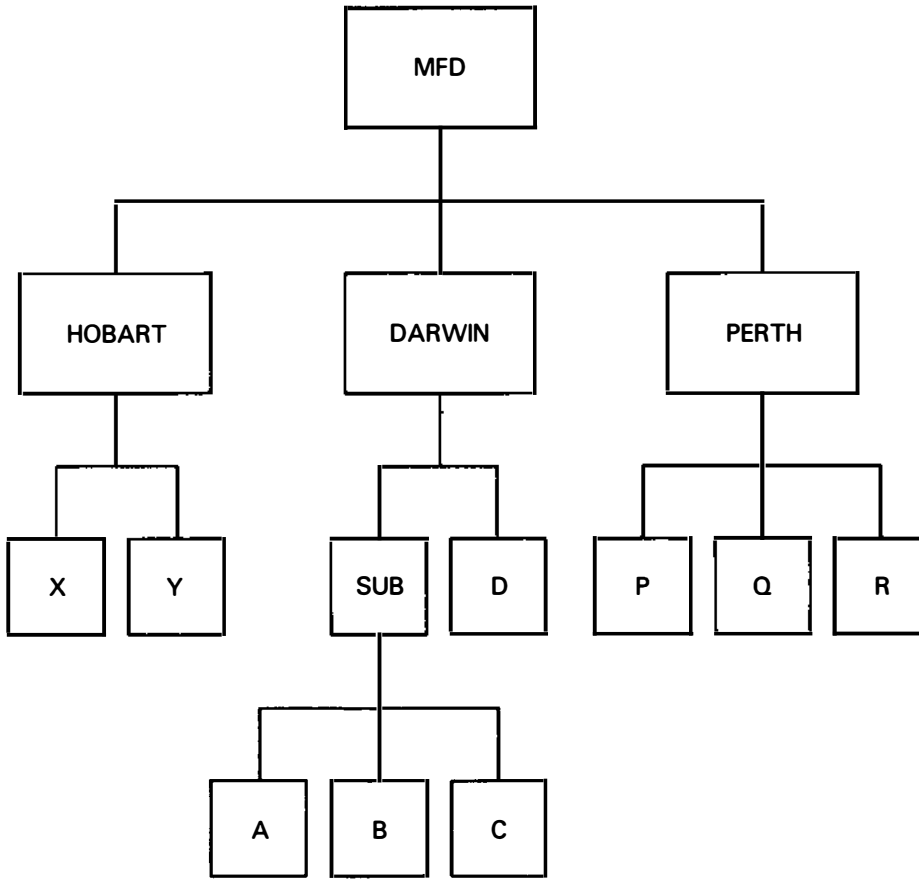
Because directories are files with no special attributes, they may list files that are in turn directories. The user may construct directory hierarchies of arbitrary depth and complexity to structure files as needed. The maximum depth of a directory hierarchy is nine levels.

Historically, Files–11 on PDP–11 systems supported a two-level directory hierarchy that relied on UIC syntax. Each UIC was associated with a user file directory (UFD) and was referenced by a UIC construction of the form **[nnn,mmm]**. This construction then translated to a user file directory name of **nnnmmm.DIR;1** in the master file directory (MFD).

The current file system uses a multilevel directory hierarchy. The first level below the volume's master file directory is the user file directory, but further levels in the directory structure may be defined; these are called subfile directories (SFDs). The top-level directory is generally used to represent individual system users or important facilities. As a result, MFD entries would correspond to the user names in a multiuser system.

Figure 2–18 shows the hierarchical directory structure.

Figure 2–18: Hierarchical Directory Structure



A directory specifier has the format **[name1.name2.name3. . . .]**. Each name in the list translates to a directory file name of the form **name.DIR;1** in the current directory level.

The current file system still supports the former UIC-based directory structure.

2.4.3.1 Multivolume Directory Structure

In a volume set, the MFD for all the user files on the volume set is the MFD of relative volume 1. Its entries point to UFDs located on any volume in the set. The UFD entries in turn point to files and subdirectories on any volume in the set. The MFDs of the remaining volumes in the set list only the reserved files on each volume.

2.5 Reserved Files

Any file system must maintain some data structures on the medium that are used to control the file organization. In Files–11, this data structure is kept in several files. These files are created when a new volume is initialized. They are unique in that their file identifiers are known constants.

The relative volume number used when accessing one of these files depends upon the context. The exact number of these files which is present on a particular volume may vary, but at least five must be present. None of these files can be deleted. Table 2–14 shows the nine reserved files.

Table 2–14: Reserved Files

File ID	File Name	Description
1,1	INDEXF.SYS;1	Index file. This file is the root of the entire Files–11 structure. It contains the volume's bootstrap block (or boot block) and the home block, which identifies the volume and locates the rest of the file structure. The index file also contains all of the file headers for the volume and a bitmap to control their allocation.
2,2	BITMAP.SYS;1	Storage bitmap file. This file controls the allocation of logical blocks on the volume.
3,3	BADBLK.SYS;1	Bad block file. This file contains all the known bad blocks on the volume.
4,4	000000.DIR;	Master file directory (MFD). This file forms the root of the volume's directory structure. The MFD lists the nine known files, all first-level user directories, and whatever other files the user chooses to enter.
5,5	CORIMG.SYS;1	System core image file. This file provides a file of known file identification for the operating system. Its use depends on the operating system. This file is not currently used.
6,6	VOLSET.SYS;1	Volume set list file. This file contains a list of the labels of the volumes in a tightly coupled volume set if this volume is the first relative volume of such a set.
7,7	CONTIN.SYS;1	Standard continuation file. This file contains the first segment of the portion of the multivolume file that resides on a loosely coupled volume set if this volume is part of such a set.

(continued on next page)

Table 2–14 (Cont.): Reserved Files

File ID	File Name	Description
8,8	BACKUP.SYS;1	Backup file. This file logs and controls an incremental backup system. This file is not currently used.
9,9	BADLOG.SYS;1	Pending bad block log file. This file contains a list of suspected bad blocks on the volume that have not yet been turned over to the bad block file.

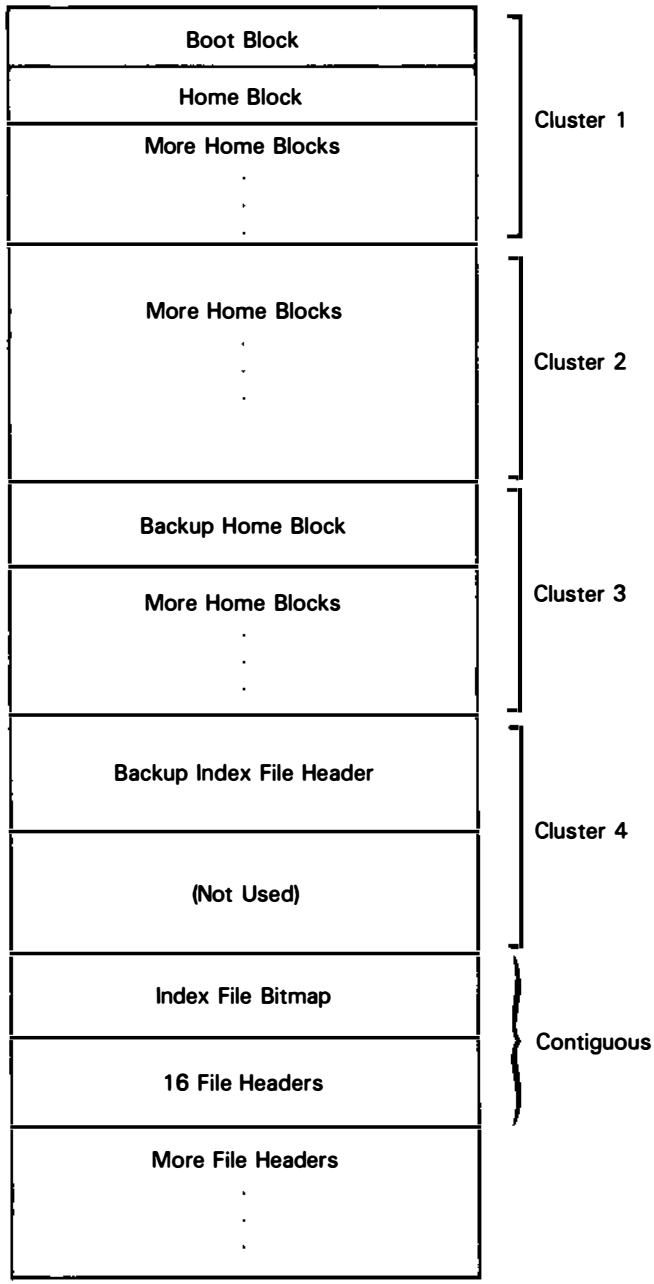
NOTE: Digital may reserve more file identifiers in the future, so users should make no assumptions about the values of user-created file identifiers.

2.5.1 Index File

The index file has file ID 1,1. It is listed in the MFD as INDEXF.SYS;1. The index file is the root of the Files–11 structure in that it provides the means for identification and initial access to a Files–11 volume. It also contains the access data for all files on the volume, including itself. This file has the record format of 512-byte fixed-length records with no carriage control.

Figure 2–19 shows the layout of the blocks in the index file. This figure assumes a storage map cluster factor greater than 2.

Figure 2–19: Layout of the First Extent of the Index File



ZK-9584-HC

2.5.1.1 Bootstrap Block

Virtual block 1 of the index file is the volume's **bootstrap block**, or **boot block**. It is almost always mapped to logical block 0 of the volume. If the volume is the system device of an operating system, the boot block contains an operating-system-dependent program that reads the operating system into memory when the boot block is read and executed by a machine's hardware bootstrap.

If the volume is not a system device, the boot block contains a small program that outputs a message on the system console to inform the operator to that effect. If block 0 of a volume is bad, VBN 1 of the index file can be mapped to some other block. However, the volume cannot be used as a system volume.

2.5.1.2 Home Block

Virtual block 2 of the index file is the volume's **home block**. The home block identifies the volume as a Files–11 volume, establishes the specific identity of the volume, and serves as the entry point into the volume's file structure. The home block is recognized as a home block by the presence of checksums in known places and by the presence of predictable values in certain locations.

The home block is located on the first good block of the home block search sequence. The formula of the search sequence is

$$1 + n * delta$$

where n is a positive integer (such as 0, 1, 2, 3, . . .).

The home block search delta is calculated from the geometry of the volume. If the volume is viewed as a three-dimensional space, the search sequence approximately travels down the body diagonal of the space. Since volume failures tend to occur across one dimension, this algorithm minimizes the chance of a single failure destroying both home blocks of the search sequence.

The volume geometry is expressed in sectors (s), tracks or surfaces (t), and cylinders (c), and the search delta is calculated according to the following rules, to handle the cases where one or two dimensions of the volume have a size of 1.

Geometry	Delta
s x 1 x 1	1
1 x t x 1	1
1 x 1 x c	1
s x t x 1	s + 1
s x 1 x c	s + 1
1 x t x c	t + 1
s x t x c	(t + 1) x s + 1

In most cases, however, the home block is located on LBN 1.

The fields of the home block are shown in Figure 2–20 and are described in Table 2–15. All copies of the volume's home block contain the same data, with the exception of the fields containing the block's VBN and LBN.

Figure 2–20: Format of the Home Block

HM2$L_HOMELBN			0
HM2$L_ALHOMELBN			4
HM2$L_ALTIDXLBN			8
HM2$W_CLUSTER	HM2$W_STRUCLEV		12
HM2$W_ALHOMEVBN	HM2$W_HOMEVBN		16
HM2$W_IBMAPVBN	HM2$W_ALTIDXVBN		20
HM2$L_IBMAPLBN			24
HM2$L_MAXFILES			28
HM2$W_RESFILES	HM2$W_IBMAPSIZE		32
HM2$W_RVN	HM2$W_DEVTYPE		36
HM2$W_VOLCHAR	HM2$W_SETCOUNT		40
HM2$L_VOLOWNER			44
reserved			48
HM2$W_FILEPROT	HM2$W_PROTECT		52
HM2$W_CHECKSUM1	reserved		56
HM2$Q_CREDATE			60
HM2$W_EXTEND	HM2$B_LRU_LIM	HM2$B_WINDOW	68

(continued on next page)

Figure 2–20 (Cont.): Format of the Home Block

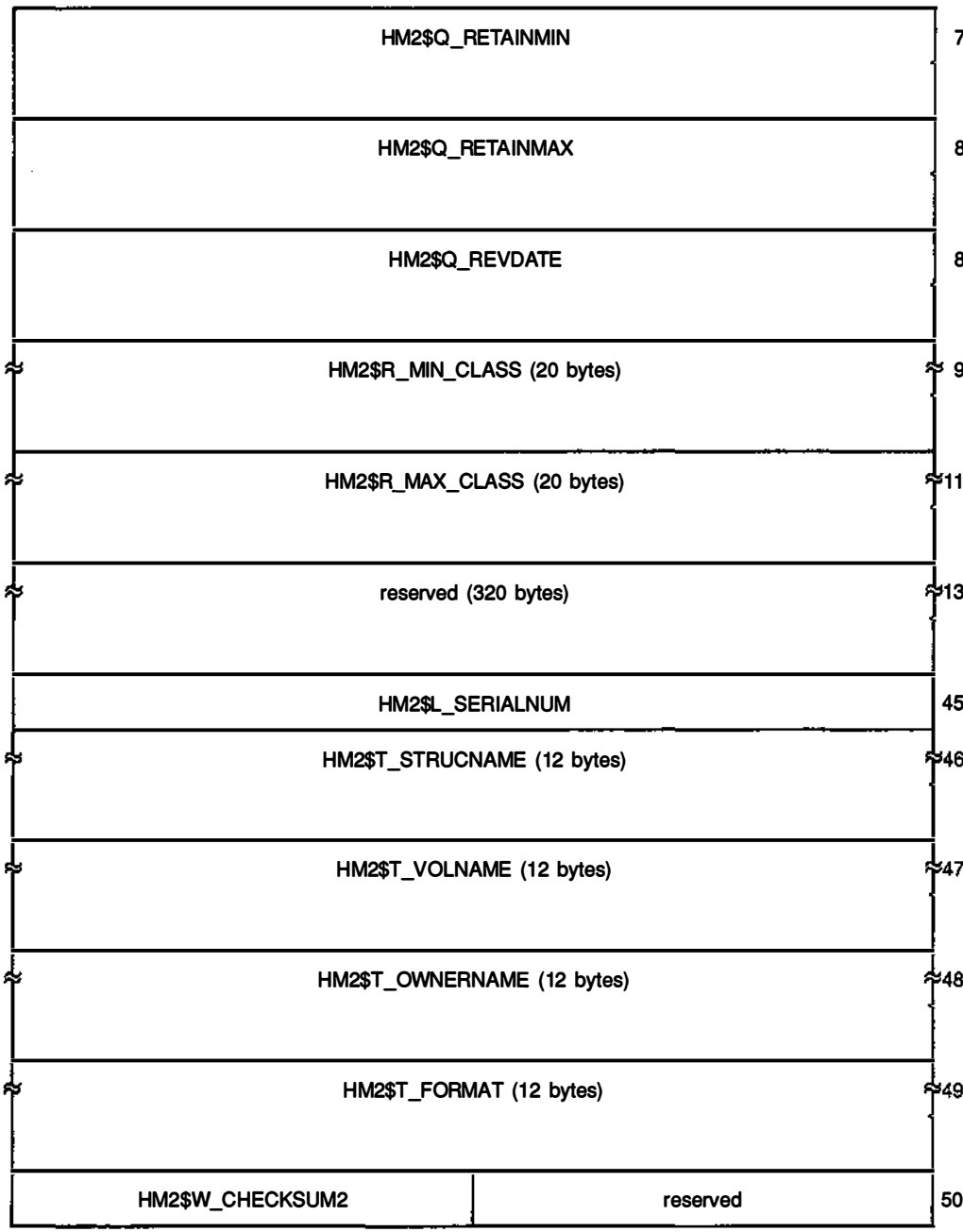


Table 2–15: Contents of the Home Block

Field Name	Description
HM2$L_HOMELBN	Home block LBN. This longword contains the logical block number of this particular copy of the home block. This value must be nonzero for a valid home block.
HM2$L_ALHOMELBN	Alternate home block LBN. This longword contains the LBN of the volume's secondary home block. When scanning the home block search sequence, the user may determine whether the block read is the primary or the secondary home block by comparing the HM2$HOMELBN field with this field. This value must be nonzero for a valid home block.
HM2$L_ALTIDXLBN	Backup index file header LBN. This longword contains the logical block on which the backup index file header is located. This value must be nonzero for a valid home block.
HM2$W_STRUCLEV	Structure level and version. The volume structure level and version is used to identify different versions of Files–11 when they affect the structure of all parts of the volume except the file header. Because the structure level word identifies the version of Files–11 that created this particular volume, upward compatibility of file structures as Files–11 evolves is assured. The current structure level of Files–11 is level 2, so the high byte of this field must contain the value 2. The low byte contains the version number, which must be greater than or equal to 1. The version number will be incremented when compatible additions are made to the Files–11 structure. The current version is version 1 of structure level 2.
HM2$W_CLUSTER	Storage bitmap cluster factor. This word contains the cluster factor used in the storage bitmap file. The cluster factor is the number of blocks represented by each bit in the storage bitmap. This value is also called the **volume cluster factor**. It must be nonzero for a valid home block.

(continued on next page)

Table 2–15 (Cont.): Contents of the Home Block

Field Name	Description
HM2$W_HOMEVBN	Home block VBN. This word contains the virtual block that this particular copy of the home block occupies in the index file. This value must be nonzero for a valid home block.
HM2$W_ALHOMEVBN	Backup home block VBN. This word contains the virtual block number that the cluster containing the secondary home block occupies in the index file. The contents of this word is calculated with the formula $v * 2 + 1$, where v is the storage map cluster factor.
HM2$W_ALTIDXVBN	Backup index file header VBN. This word contains the virtual block number that the backup index file header occupies in the index file. The contents of this word is calculated with the formula $v * 3 + 1$, where v is the storage map cluster factor.
HM2$W_IBMAPVBN	Index file bitmap VBN. This word contains the starting virtual block number of the index file bitmap. The contents of this word is calculated with the formula $v * 4 + 1$, where v is the storage map cluster factor.
HM2$L_IBMAPLBN	Index file bitmap LBN. This longword contains the starting logical block address of the index file bitmap. Once the home block of a volume has been found, this value provides access to the rest of the index file and to the volume. This value must be nonzero for a valid home block.
HM2$L_MAXFILES	Maximum number of files. This longword contains the maximum number of files that may be present on the volume at any time. This value must be greater than the contents of HM2$W_RESFILES for the home block to be valid. The maximum number of files cannot exceed $2^{24} - 1$.
HM2$W_IBMAPSIZE	Index file bitmap size. This 16-bit word contains the number of blocks that make up the index file bitmap. This value must be nonzero for a valid home block.
HM2$W_RESFILES	Number of reserved files. This word contains the number of reserved files on the volume. The file sequence number of each reserved file is always equal to its file number. Reserved files may not be deleted, and at least five must be present on a volume. To be valid, this word cannot contain a value less than 5.
HM2$W_DEVTYPE	Disk device type. This word is an index identifying the type of disk that contains this volume. It is currently not used and always contains a value of 0.

(continued on next page)

Table 2–15 (Cont.): Contents of the Home Block

Field Name	Description
HM2$W_RVN	Relative volume number. This word contains the relative volume number that this volume has been assigned in a volume set. If the volume is not part of a volume set, then this word contains a value of 0.
HM2$W_SETCOUNT	Number of volumes. This word contains the total number of volumes in a tightly coupled volume set if this volume is the first volume of the set (the HM2$W_RVN field contains 1). In a loosely coupled volume set, this word contains a value of 0.
HM2$W_VOLCHAR	Volume characteristics. This word contains bits that provide additional control over access to the volume. The following bits are defined: HM2$V_READCHECK — Set if read-check operations are to be performed. All read operations on the file, both for data and for file structure, are verified with a read-compare operation to ensure data integrity. HM2$V_WRITCHECK — Set if write-check operations are to be performed. All write operations on the file, both for data and for file structure, are performed with a read-compare operation to ensure data integrity. HM2$V_ERASE — Set if all files on the volume are to be erased or overwritten when they are deleted. HM2$V_NOHIGHWATER — Set if highwater mark enforcement is to be disabled on the volume.
HM2$L_VOLOWNER	Volume owner. This longword contains the binary identification code of the owner of the volume. The format is the same as that of the file owner stored in the file header.

(continued on next page)

Table 2–15 (Cont.): Contents of the Home Block

Field Name	Description
HM2$W_PROTECT	Volume protection code. This word contains the protection code for the entire volume. All operations on all files on the volume must pass both the volume protection and the file protection checks to be allowed. Just as in file protection, accessors to the volume are categorized into system, owner, group and world. Each category is controlled by the standard 4-bit field, which is encoded in the following manner: Bit <0> If set, files cannot be read. Bit <1> If set, existing files cannot be written to (modified). Bit <2> If set, files cannot be created. Bit <3> If set, files cannot be deleted.
HM2$W_FILEPROT	Default file protection. This word contains the file protection that is assigned to files created on this volume if no file protection is specified by the user or the operating system. This field is not currently supported.
HM2$W_CHECKSUM1	First checksum. This word is an additive checksum of all preceding entries in the home block. It is computed by the same sort of algorithm as the file header checksum.
HM2$Q_CREDATE	Volume creation date. This quadword contains the date and time that the volume was initialized. It has the same binary format as the file header.
HM2$B_WINDOW	Default window size. This byte contains the number of retrieval pointers that are used for the window (in core file access data) when files are accessed on the volume, if this value is not specified by the user.
HM2$B_LRU_LIM	Directory preaccess limit. This byte contains a count of the number of directories to be stored in the file system's directory access cache. It is also an estimate of the number of concurrent users of the volume.
HM2$W_EXTEND	Default file extend. This word contains the number of blocks allocated to a file when a user extends the file and asks for the system default value for allocation.

(continued on next page)

Table 2–15 (Cont.): Contents of the Home Block

Field Name	Description
HM2$Q_RETAINMIN	Minimum file retention period. This field contains the minimum length of time that a file will be retained by a file expiration system after that file is last accessed. Its value is expressed in the standard delta time format (minus the time in tenths of microseconds).
HM2$Q_RETAINMAX	Maximum file retention period. This field contains the maximum length of time that a file will be retained by a file expiration system after that file is last accessed. Its value is also expressed in the standard delta time format. The minimum and maximum retention fields are used together in this way: when a file is accessed, if the sum of the current time plus the minimum retention period exceeds the current expiration date of the file, then the file's expiration date is reset to the sum of the current time plus the maximum retention period. In other words, how often the expiration date of a frequently accessed file is updated is determined by the difference between the minimum and the maximum retention periods.
HM2$Q_REVDATE	Volume revision date. This field contains the date and time at which the last significant modification, such as copying during a full backup, was made to the volume.
HM2$R_MIN_CLASS	Minimum security class. This field contains a classification mask that represents the minimum secrecy and integrity classification of files that may be created on this volume. The structure of this field is the same as that of the secrecy classification mask (FH2$R_CLASS_PROT) in the header area.
HM2$R_MAX_CLASS	Maximum security class. This field contains a classification mask that represents the maximum secrecy and integrity classification of files that may be created on this volume. The structure of this field is the same as that of the secrecy classification mask (FH2$R_CLASS_PROT) in the header area.
HM2$L_SERIALNUM	Media serial number. This field contains the binary serial number of the physical medium on which the volume is located.

(continued on next page)

Table 2–15 (Cont.): Contents of the Home Block

Field Name	Description
HM2$T_STRUCNAME	Structure name. This area contains the name of the volume set (in ASCII form) to which this volume belongs, padded to 12 bytes with spaces. The characters must be ASCII characters that can be printed (that is, no control or delete characters). Moreover, using only alphanumerics is also recommended to avoid conflicts with command languages. This field may not be null. If this volume is not a member of a volume set, this area is filled with spaces.
HM2$T_VOLNAME	Volume name. This area contains the volume label in ASCII form. It is padded to 12 bytes with spaces. The characters must be ASCII characters that can be printed (that is, no control or delete characters). Moreover, using only alphanumerics is also recommended to avoid conflicts with command languages. This field may not be null. If the volume is a member of a shadow set, the name is the same across all the members.
HM2$T_OWNERNAME	Volume owner. This area contains an ASCII string identifying the owner of the volume. The area is padded to 12 bytes with trailing spaces.
HM2$T_FORMAT	Format type. This field contains the ASCII string "DECFILE11B" padded to 12 bytes with spaces. It identifies the volume as being of Files–11 format, structure level 2.
HM2$W_CHECKSUM2	Second checksum. This word is the last word of the home block. It contains an additive checksum of the preceding 255 words of the home block, calculated by the same algorithm used to calculate the end checksum of the header area.

2.5.1.3 Cluster Filler

If the cluster factor v of the volume is greater than 1, then the next $v*2-2$ blocks of the index file are copies of the home block used to fill out the first two clusters of the index file. Note that, for cluster factors greater than 1, this method results in a wasted disk cluster. The benefit of this technique is a much simpler rule for finding the VBN of parts of the index file.

2.5.1.4 Backup Home Block

The **backup home block** is a copy of the home block that is located farther along the home block search sequence. It permits the volume to be used even if the primary home block is destroyed.

In general, the backup home block should be allocated on the second good block of the search sequence. If it is not, then no preceding block of the sequence can be available for allocation. Otherwise, a malicious user could construct a counterfeit index file which would be used if the primary home block were corrupted.

The cluster which contains the backup home block is mapped into the index file as virtual blocks $v * 2 + 1$ through $v * 3$, where v is the volume cluster factor.

The backup home block may be located anywhere within this cluster because there is no definite relationship between the cluster factor and the volume's track and cylinder boundaries. The entire cluster is therefore filled out with copies of the home block. The file system must be able to allocate two good home blocks on the home block search sequence. The blocks in the sequence preceding the two home blocks that are not used for home blocks must be marked both bad and allocated, especially LBN 1.

2.5.1.5 Backup Index File Header

The next cluster of the index file contains the **backup index file header** so data on the volume can be recovered if the index file header is corrupted. The cluster occupies virtual blocks $v * 3 + 1$ through $v * 4$, where v is the volume cluster factor.

The LBN of the backup index file header is stored in location HM2$L_ALTIDXLBN in the home block. The backup index file header occupies the first block of this cluster. The remaining blocks are not used, so their contents are undefined.

2.5.1.6 Index File Bitmap

The **index file bitmap** is used to control the allocation of file numbers and file headers. It is simply a bit string of length n, where n is the maximum number of files allowed on the volume. This value is stored in the HM2$L_MAXFILES field in the home block.

The bitmap spans as many blocks as needed to map the allocation of the files on the volume. This number is the maximum number of files divided by 4096 and rounded up. The number of blocks in the bitmap is contained in the HM2$W_IBMAPSIZE field of the home block.

The bits in the index file bitmap are numbered sequentially from 0 to $n - 1$ from right to left in each byte, and in order of increasing byte address. Bit j is used to represent file number $j + 1$. If the bit is set (or 1), then that file number is in use; if the bit is clear (or 0), then that file number is not in use and may be assigned to a newly created file. The index file bitmap is not used to determine whether a header is valid when accessing an existing file; the validation is done using the contents of the header itself.

The index file bitmap starts at virtual block $v*4+1$ of the index file and continues through VBN $v * 4 + m$, where m is the number of blocks in the bitmap, and v is the storage map cluster factor. It is located at the logical block indicated by the HM2$L_IBMAPLBN field in the home block.

2.5.1.7 File Headers

The rest of the index file contains all the file headers for the volume. The first sixteen file headers (for file numbers 1 to 16) are logically contiguous with the index file bitmap to make them easy to locate, but the rest may be suitably allocated wherever the file system decides. Thus, the first sixteen file headers may be located from the information in the home block (HM2$W_IBMAPSIZE and HM2$L_IBMAPLBN) while the rest must be located through the mapping data in the index file header. The file header for file number n is located at virtual block $v * 4 + m + n$, where m is the number of blocks in the index file bitmap and v is the storage map cluster factor.

The end-of-file (EOF) mark for the index file is located at or beyond the last file header ever used. All header blocks located before the end-of-file mark must be validated when they are used to create a new file. If the block does not contain a valid file header, it is allocated for a new header. The new header is assigned a file sequence number of 1 if it is the first use of this header block. Index file blocks beyond the end-of-file mark are assumed not to be valid file headers for the purpose of creating new file headers.

If the block contains a deleted file header, the new header is assigned a sequence number one higher than the header currently contained in the block.

A block containing a valid file header must never be used to create a new file, even if it is marked free in the index file bitmap. This rule prevents files from being lost if bits are dropped in the bitmap.

2.5.2 Storage Bitmap File

The storage bitmap file has file ID 2,2. It is listed in the MFD as BITMAP.SYS;1. The storage bitmap is used to control the available space on a volume. It consists of both a **storage control block**, or SCB, which contains summary information about the volume, and the **storage bitmap** itself, which lists the individual blocks that are available for allocation.

This file has the format of 512-byte fixed-length records with no carriage control. The end-of-file mark points to the last block used. The storage bitmap file must be contiguous.

2.5.2.1 Storage Control Block

Virtual block 1 of the storage bitmap is the storage control block (SCB). It contains summary information about the volume. Note that some of the features in the SCB require it to be written when the volume is mounted or dismounted. The fields of the SCB are shown in Figure 2–21 and are described in Table 2–16.

Figure 2–21: Format of the Storage Control Block

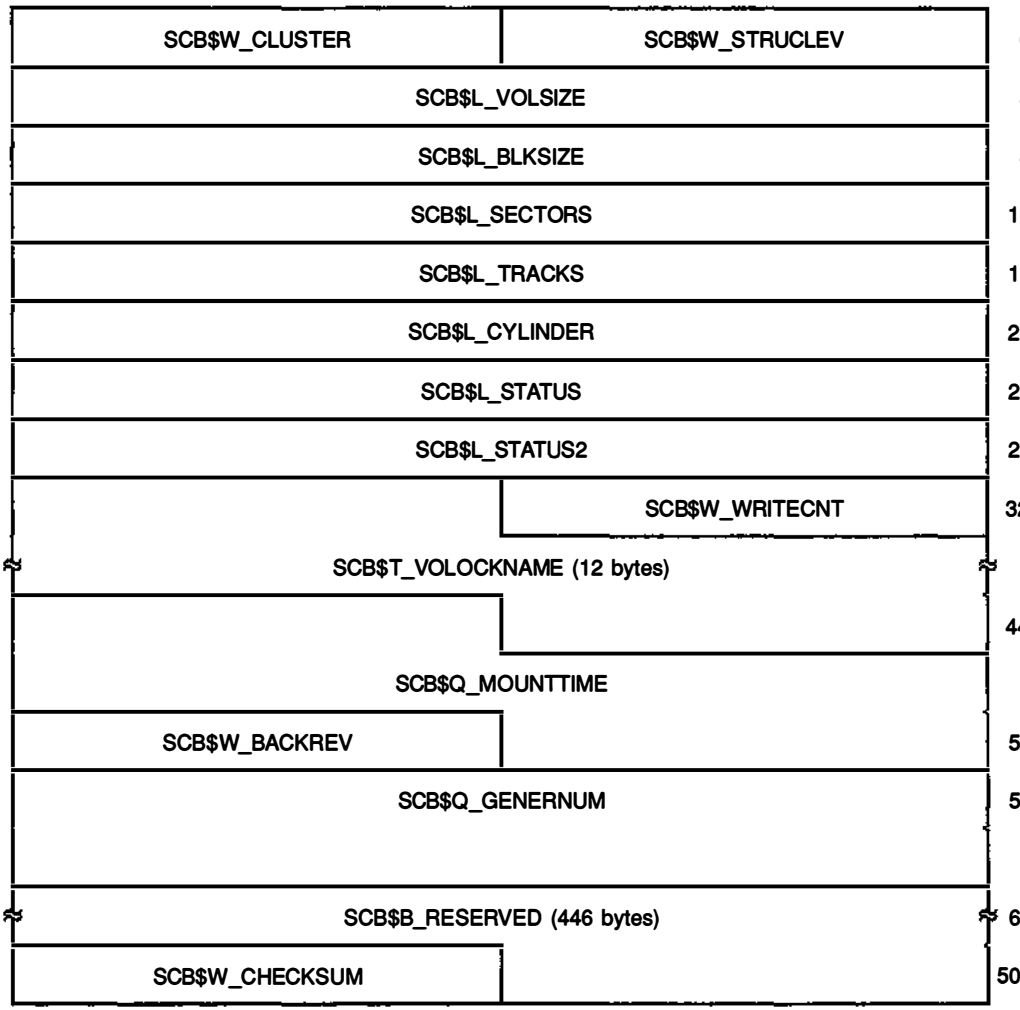


Table 2–16: Contents of the Storage Control Block

Field Name	Description
SCB$W_STRUCLEV	Storage map structure level. This word contains the structure level of the storage control block. The high byte contains the value 2 to indicate Files–11 structure level 2. The low byte contains the version number, which must be greater than or equal to 1.
SCB$W_CLUSTER	Storage map cluster factor. This word contains the storage map cluster factor of the volume. Its contents are identical to the contents of HM2$W_CLUSTER in the home block. It is placed here for convenience.
SCB$L_VOLSIZE	Volume size. This field contains the volume size expressed as logical blocks.
SCB$L_BLKSIZE	Blocking factor. This field contains the blocking factor of the volume, that is, the number of physical blocks or sectors that make up one logical block.
SCB$L_SECTORS	Sectors per track. This field contains the number of logical blocks in each track of the volume.
SCB$L_TRACKS	Tracks per cylinder. This field contains the number of tracks contained in each cylinder of the volume.
SCB$L_CYLINDER	Number of cylinders. This field contains the total number of cylinders on the volume. The preceding three fields allow space on the physical boundaries of the volume to be allocated in an optimized manner.

(continued on next page)

Table 2–16 (Cont.): Contents of the Storage Control Block

Field Name	Description
SCB$L_STATUS	Status word. This word contains the volume status flags that follow; these bits are not currently supported. SCB$V_MAPDIRTY — Set if the storage map is "dirty," or only partially updated. SCB$V_MAPALLOC — Set if the storage map is preallocated, which may result in lost blocks. SCB$V_FILALLOC — Set if file numbers are preallocated, which may result in lost header slots. SCB$V_QUODIRTY — Set if the quota file is "dirty," or only partially updated. SCB$V_HDRWRITE — Set if file headers are to be cached with write-back operations.
SCB$L_STATUS2	Secondary status. This word contains a copy of the status flags while the volume is mounted for write access. The volume flag bits match those of the SCB$L_STATUS field. These bits are not currently supported. The volume status flags are as follows: SCB$V_MAPDIRTY2 — Set if the storage map is "dirty," or only partially updated. SCB$V_MAPALLOC2 — Set if the storage map is preallocated, which may result in lost blocks. SCB$V_FILALLOC2 — Set if file numbers are preallocated, which may result in lost header slots. SCB$V_QUODIRTY2 — Set if the quota file is "dirty," or only partially updated. SCB$V_HDRWRITE2 — Set if file headers are to be cached with write-back operations.
SCB$W_WRITECNT	Writer count. This word contains the number of systems that have the volume currently mounted for write access.
SCB$T_VOLOCKNAME	Volume lock name. This word contains a unique name used as a root for file system serialization or resource synchronization on the volume.

(continued on next page)

Table 2–16 (Cont.): Contents of the Storage Control Block

Field Name	Description
SCB$Q_MOUNTTIME	Time of last mount. This field contains the date and time of the last time the volume was mounted for write access. It is expressed in the standard time format.
SCB$W_BACKREV	BACKUP revision number. This field indicates the number of times the volume has been copied using the DCL command BACKUP/IMAGE.

SCB$Q_GENERNUM	Shadow set revision number. This field is the basic shadow volume generation indicator. It has two characteristics:
	• Its value always increases.
	• It represents the time at which the most recent change in shadow set status occurred while the volume was still a member of the shadow set.
	If the two characteristics conflict, the increasing value takes priority, and the generation number is updated using the following algorithm:
	`Let CURRENT_TIME be the current time.`
	`Let OLD_GENERNUM be the generation number to be increased.`
	`Let NEW_GENERNUM be the increased generation number.`
	`If CURRENT_TIME > OLD_GENERNUM, then NEW_GENERNUM = CURRENT_TIME;`
	`otherwise NEW_GENERNUM = OLD_GENERNUM + 1.`
	The generation number is updated by writing directly to the shadow set, never to the individual members of the shadow set. It is updated just before the entire shadow set is dismounted or after processing is completed when the membership of the shadow set changes. A change in membership occurs when a volume is added, a volume is removed because of hardware failure, or a volume is removed with a user command. Note that an added volume will receive the updated generation number, but a removed volume will not because the generation number is written to the current membership only after the addition or the removal has been completed.

(continued on next page)

Table 2–16 (Cont.): Contents of the Storage Control Block

Field Name	Description
SCB$B_RESERVED	Reserved.
SCB$W_CHECKSUM	End checksum. This word contains the block checksum. It is calculated using the same algorithm as the end checksum of the header area.

2.5.2.2 Storage Bitmap

Virtual blocks 2 through $n + 1$ are the storage bitmap itself. It is best viewed as a bit string of length m, numbered from 0 to $m - 1$, where m is the total number of clusters on the volume rounded up to the next multiple of 4096.

Each cluster contains v logical blocks, where v is the storage map cluster factor (also referred to as the volume cluster factor) contained in the field in the home block. The bits are addressed in the usual manner (packed right to left in sequentially numbered bytes).

Since each virtual block holds 4096 bits, n blocks (where $n = \frac{m}{4096}$) are used to hold the bitmap. Bit j of the bitmap represents logical blocks $j * v$ through $j * v + v - 1$ of the volume. If the bit is set, the blocks are free; if clear, the blocks are allocated. The last k bits of the bitmap are always clear, where k is the difference between the true size of the volume and m, the length of the bitmap.

Rounding the storage map file up to the next multiple of the volume cluster factor may result in some unused blocks at the end of the file. The end-of-file mark points to the last block used.

2.5.3 Bad Block File

The bad block file has file ID 3,3. It is listed in the MFD as BADBLK.SYS;1. The bad block file is simply a file containing all the known blocks on the volume that cannot reliably store data. This file has the record format of 512-byte fixed-length records with no carriage control.

On disks containing a bad block descriptor, the last track of the volume comprises the first several clusters of the bad block file. This rule ensures that the bad block data is available to software in a file-structured manner and is preserved when the volume is initialized again.

The end-of-file mark is placed during volume initialization at the end of the bad blocks found during the initialization. At all times, the end-of-file mark must point past the bad block descriptor data.

2.5.3.1 Manufacturer's Bad Block Descriptor

Many Digital-supplied disks that predate Digital Storage Architecture (DSA) disks have a manufacturer-supplied format that lists on the volume's last (highest) track all the known bad blocks or sectors. It is written in 16-bit format. Disks of this type include the RK06, RK07, RP06, and the RM03.

The fields of a manufacturer-supplied bad block descriptor are shown in Figure 2–22 and are described in Table 2–17.

Figure 2–22: Format of the Manufacturer-Supplied Bad Block Descriptor

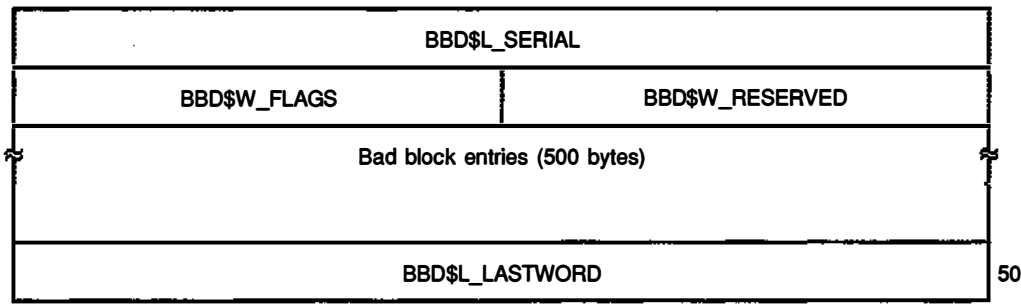


Table 2–17: Contents of a Manufacturer-Supplied Bad Block Descriptor

Field Name	Description
BBD$L_SERIAL	Serial number of the disk.
BBD$W_RESERVED	Reserved area.
BBD$W_FLAGS	Status flags. This field contains a value of 0 for normal use. A nonzero value is used to identify maintenance disks, which should never be initialized.

(continued on next page)

Table 2–17 (Cont.): Contents of a Manufacturer-Supplied Bad Block Descriptor

Field Name	Description
BBD$L_BADBLOCK	Bad block entry. This longword is an individual bad block entry, which identifies a defective block on the disk. The start of the bad block entries is pointed to by the symbol BBD$C_DESCRIPT. A list of bad block entries may occupy up to 500 bytes; that is, only 126 bad block entries may be recorded. After the last bad block has been listed, the rest of the field is padded with all 1s, which identifies the end of the bad block list. The format of an individual bad block entry is shown in Figure 2–23.
BBD$L_LASTWORD	Last longword of block. This field contains all 1s to signal the end of the bad block descriptor to the file system.

Figure 2–23 shows the format of an individual bad block entry. Table 2–18 describes the bits contained in the BBD$L_BADBLOCK field.

Figure 2–23: Format of a Bad Block Entry (BBD$L_BADBLOCK)

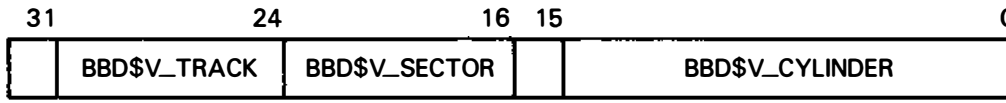


Table 2–18: Contents of a Bad Block Entry

Bit	Meaning
BBD$V_CYLINDER	Cylinder number of the bad block. This field is 15 bits long and starts at bit 0.
BBD$V_SECTOR	Sector number of the bad block. This field is 8 bits long and starts at bit 16.
BBD$V_TRACK	Track number of the bad block. This field is 7 bits long and starts at bit 24.

All the sectors on the last track that contain bad block data are written in the same format as any other track on the disk; that is, they have the same preamble, gaps, error correction code (ECC), and postamble. The even-numbered sectors of

the highest ten sectors (that is, sectors 0, 2, 4, 6, and 8) are available for the manufacturer's bad block descriptor.

The rest of the sectors on the last track (sectors 10 to 21) contain the software bad block descriptor. These sectors have the same format as the manufacturer's bad block descriptor except that the area for the bad block entries contains all 1s, which indicates that no bad sectors are listed. This area is where the system software may list the sectors that have become defective while in use.

2.5.3.2 Software Bad Block Descriptor

For disks that do not have factory last-track bad block data and are not DSA disks, a software-generated bad block map is supplied. It is always located on the last good block of the volume. There must be at least one reliable block in the last 256 blocks of the volume for this bad block map to be generated.

The fields of a bad block descriptor are illustrated in Figure 2–24 and are described in Table 2–19.

Figure 2–24: Format of a Software Bad Block Descriptor

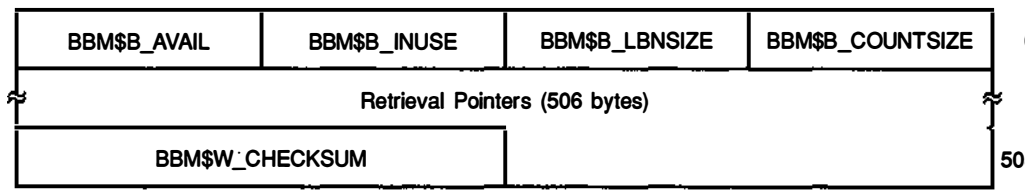


Table 2–19: Contents of a Software Bad Block Descriptor

Field Name	Description
BBM$B_COUNTSIZE	Count size. This field contains the retrieval pointer count field size, which must always contain a value of 1.
BBM$B_LBNSIZE	Logical block number. This field contains the retrieval pointer LBN field size, which must always contain a value of 3.
BBM$B_INUSE	Map words in use. This field contains the number of retrieval words that contain bad block data.

(continued on next page)

Table 2–19 (Cont.): Contents of a Software Bad Block Descriptor

Field Name	Description
BBM$B_AVAIL	Map words available. This field contains the number of retrieval words that are available to bad block data.
BBM$W_CHECKSUM	End checksum.

Each bad block descriptor retrieval pointer is four bytes long. The fields of a retrieval pointer are shown in Figure 2–25 and are described in Table 2–20.

Figure 2–25: Format of a Bad Block Descriptor Retrieval Pointer

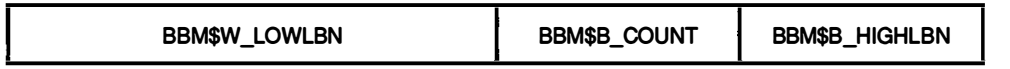


Table 2–20: Contents of a Bad Block Descriptor Retrieval Pointer

Field Name	Description
BBM$B_HIGHLBN	High-order LBN. This field contains the high-order bits of the 24-bit LBN.
BBM$B_COUNT	Block count. This field contains the count field (in excess 1 format).
BBM$W_LOWLBN	Low-order LBN. This field contains the low 16 bits of the 24-bit LBN field.

2.5.3.3 Bad Block Processing on DSA Disks

Disks conforming to the Digital Storage Architecture (DSA) format have no visible bad blocks. Disks of this type include the RA60, RA80, RA87, RA81, and RA90. Instead, the hardware and the disk class driver (DUDRIVER) produce a logically contiguous range of good blocks. If a block in the user area of the disk becomes defective, further accesses to that block are revectored to a nearby spare good block. A percentage of blocks are reserved for revectoring when the disk is formatted.

There are two types of block replacement:

- **Host-initiated**—The operating system implements the replacement algorithm because the disk controller does not have sufficient memory.
- **Controller-initiated**—A disk controller with sufficient memory (such as an HSC) implements the replacement algorithm.

The list of replacement blocks is kept in the **replacement and caching table** (RCT). The RCT provides the control structures and extra storage used during automatic block replacement operations. The RCT also keeps a list of defective blocks and replacement blocks currently in use. In a sense, the RCT replaces the manufacturer's bad block list because the factory enters any defective blocks in the RCT before shipment. As a result, it is never necessary to run the Bad Block Locator Utility (BAD) on DSA disks. Also, the file BADBLK.SYS;1 is empty.

Any inconsistencies in the RCT will cause the disk to be write-locked automatically. The RCT can become invalid in the following two ways:

- All the replacement blocks may be in use. In this case, an entry is made in the error logger, and the disk is mounted for read access only.
- The blocks of the RCT itself cannot be revectored, so the RCT is recorded multiple times on the disk. The locations are maintained by the disk controller. Each copy of the RCT is sequentially read from or updated during read or write operations. If and only if every copy is defective, the volume is automatically protected against write access.

2.5.4 Master File Directory

The master file directory has file ID 4,4. It is listed in the MFD (itself) as 000000.DIR;1. The MFD is the root of the volume's directory structure. It lists the reserved files plus entries for all top-level user file directories (UFDs). It also contains whatever files the user chooses to enter.

The format of the MFD is the same as that of all directory files. The format of directory files is covered in Section 2.4.1.

2.5.5 Core Image File

The core image file has file ID 5,5. It is listed in the MFD as CORIMG.SYS;1. Its use depends on the operating system. In general, it provides a file of known file ID for the use of the operating system (as a swap area, for example). This file has the record format of 512-byte fixed-length records with no carriage control. The end-of-file mark points to the physical end of the file.

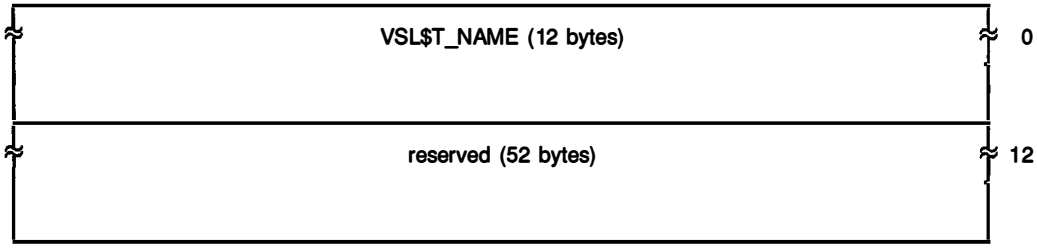
2.5.6 Volume Set List File

The volume set list file has file ID 6,6. It is listed in the MFD as VOLSET.SYS;1. It is used only on relative volume 1 of a tightly coupled volume set. It contains a list of the volume labels of the volumes contained in the volume set.

The format of this file is 64-byte fixed-length records with implied carriage control. The first 12 bytes of record 1 contain the volume set name. The first 12 bytes of record n contain the volume label of relative volume $n-1$. The remaining 52 bytes of each record are reserved.

Figure 2–26 shows the format of the volume set list file.

Figure 2–26: Format of the Volume Set List File



2.5.7 Continuation File

The standard continuation file has file ID 7,7. It is listed in the MFD as CONTIN.SYS;1. It is used as the extension file ID when a file crosses from one volume of a loosely coupled volume set to another.

The purpose of this reserved file ID is to allow a multivolume file to be written sequentially with only one volume mounted at a time. Ordinarily, when a file is extended onto another volume, the new header must be created first so that the new file ID can be obtained before the extension linkage in the current header can be written. The use of this reserved file ID allows the extension linkage to be written with a known constant before the next volume is even on line.

2.5.8 Backup Journal File

The backup journal file, also called the backup log file, has file ID 8,8. It is listed in the MFD as BACKUP.SYS;1. This file contains a history of volume and incremental backups performed on the volume. This file is not currently used.

2.5.9 Pending Bad Block Log File

The pending bad block log file has file ID 9,9. It is listed in the MFD as BADLOG.SYS;1. This file contains a list that identifies the suspected bad blocks on the volume that are not currently contained in the volume's bad block file. The format of this file is 16-byte fixed-length records.

Each record in the file represents one bad block. The format of each record is shown in Figure 2–27 and is described in Table 2–21.

For more information about how this file is used in bad block processing, see Section 5.4.8.

Figure 2–27: Format of a Pending Bad Block Log Record

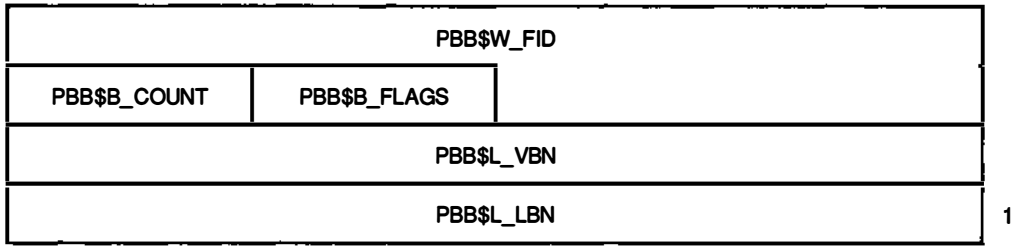


Table 2–21: Contents of a Pending Bad Block Log Record

Field Name	Description
PBB$W_FID	File ID of the file that contains the bad block.
PBB$B_FLAGS	Flags. The following flag bits are defined: PBB$V_READERR Set if a read error has occurred on this block. PBB$V_WRITERR Set if a write error has occurred on this block.
PBB$B_COUNT	Error count.
PBB$L_VBN	Virtual block number. This field contains the VBN of the bad block of the file.
PBB$L_LBN	Logical block number. This field contains the LBN of the bad block of the file.

Chapter 3
Volume Structure Processing

. . . creators of odd volumes.
Charles Lamb

This is one of those cases in which the imagination is baffled by the facts.
Winston Churchill

Outline

3.1 Introduction

This chapter describes the file system operations and data structures involved in volume structure processing.

Volume structure processing describes a broad spectrum of activities, ranging from making the disk available to the user to accessing the data on the disk. It also includes how both the in-memory and the on-disk structures are manipulated.

Part of making the disk available to the user involves initializing, mounting, and dismounting the disk. Disk initialization is covered in Section 3.2. Section 3.3 describes the Mount procedure and the primary data structures of the I/O database. Among these structures are the ACP queue block, the file control block, the relative volume table, the volume control block, and the window control block. Rebuilding the bitmap and the disk quota files is also covered in this section. The Dismount procedure is described in Section 3.4.

3.2 Initializing the Volume

Before files or data can be written to a disk volume for the first time, the volume must be initialized. The DCL command INITIALIZE is used to format data structures and to write a label to the volume. In general, the INITIALIZE command invalidates all existing data (if any) on the volume and creates a new file structure. One of the most important tasks is to place and write the volume's reserved files.

Initializing involves the following main tasks:

- Gathering and checking the preliminary parameters
- Processing the bad blocks on the disk
- Performing a Data Security Erase (DSE), should one be requested
- Placing the reserved files on the volume
- Creating the storage bitmap file
- Initializing the index file
- Creating the master file directory

Most of the modules pertaining to initializing a disk are located in the INIT facility; some are located in the MOUNT facility. The main program is located in the INIVOL module in the INIT facility, and the routine INIT_VOLUME in the INIVOL module contains most of the initialization logic.

3.2.1 Checking the Preliminary Parameters

Certain preliminary checking and processing needs to be done before the volume can actually be initialized. To begin with, the $GETJPI system service obtains the UIC of the process from which the INITIALIZE command was issued. The INITIALIZE command line is then parsed to acquire the user's input, including the device and volume name, and the results are written to a global data area.

After the command line has been parsed, the device is allocated, and a channel is assigned to it. If the user has specified a logical name on the command line, it is translated. The device characteristics are obtained with the $GETDVI system service, and the device is checked to ensure that it is a file-oriented device.

The routine INIT_DISK in the module INIDSK is then called. It defines an internal allocation table in memory containing an entry for the necessary reserved structures. The table actually consists of two parallel tables: one stores the size of allocated areas and the other stores the LBN of each area.

The volume valid bit in the unit control block (UCB$V_VALID) is set, indicating to the operating system that the volume is valid. A pack acknowledgment function (IO$_PACKACK) is then issued to enable I/O to the volume because it is not mounted.

The privilege mask of the process that issued the INITIALIZE command is checked for VOLPRO privilege. If the process does not have that privilege, the original home block of the volume is read to find the UIC of the volume owner, which is contained in the HM2$L_VOLOWNER field. The UIC must either be zero, indicating that the volume is not owned, or it must match the UIC of the process that issued the INITIALIZE command.

All the volume parameters—including protection, extension size, window size, and the number of headers—that were not specified on the INITIALIZE command line are established from the system and group defaults. These parameters are verified against the volume size and characteristics.

The volume size is calculated and rounded up to the next cluster boundary. The maximum number of files can either be specified with the INITIALIZE /MAXIMUM_FILES command or defaulted from the volume size and cluster factor.

The minimum number of preallocated headers is specified with the INITIALIZE /HEADERS command. The maximum number of headers is 65,500, and the default is 16.

The initial position of the index file is determined based on user input. The index file can be placed at the beginning, the middle, or the end of a disk. It can also start at a specific LBN. If the user did not specify the /INDEX qualifier on the INITIALIZE command line, the default position is the middle of the disk, which minimizes the disk seek time.

3.2.2 Formatting the Disk

The Initialize Utility is capable of formatting floppy disks on RX02, RX33, and RX23 disk drives. If a /DENSITY qualifier is specified on the command line, an IO$_FORMAT function is issued with an appropriate density parameter to cause the disk controller to format the disk.

3.2.3 Processing Software Bad Blocks

After the preliminary checking is completed, the routine INIT_BADBLOCK performs bad block processing to see in which blocks data cannot be written. DSA disks with replacement and caching tables (RCT) do not have visible bad blocks, so they are not processed. Disks that have fewer than 4096 blocks are also not processed by default.

If the /VERIFIED qualifier was specified on the command line, INIT_BADBLOCK first establishes whether the volume has factory last-track bad block data or software bad block data and then calls either the GET_FACTBAD or the GET_SOFTBAD routine.

If the disk has a manufacturer-supplied format, routine GET_FACTBAD marks the entire last track of the disk bad (or invalid) to prevent the software from using it. The last track of the disk contains bad block data written by the factory-formatting process. The data consists of two sections:

- The first block
- The first good block after sector 10

The purpose of this data is to record both factory- and software-detected bad block data. The list containing the factory bad block data is written in the first five even-numbered sectors (sectors 0, 2, 4, 6, and 8) of the last track of the disk. The software bad block data can be written in sectors 10 through 21. For more information on the manufacturer-supplied disk format, see Section 2.5.3.1.

After a good copy of each of the bad block lists has been found, all the bad block entries in them are processed. Descriptors containing the cylinder, sector, and track numbers of the bad blocks are created.

If the disk does not have a manufacturer-supplied format and is not a DSA disk, the user must first run the Bad Block Locator Utility (BAD) on the disk to detect defective blocks. The routine GET_SOFTBAD processes the data left by this bad block scan. It searches backward from the end of the volume to find the bad block data. When a valid bad block descriptor is found, its LBN is entered in the bad block list. Then its contents are processed and entered in the bad block list.

If the /NOVERIFIED qualifier was specified on the INITIALIZE command line, the existing bad block data on the disk is temporarily ignored until the difference between the disk size and the cluster blocking factor is calculated. Blocks that form the partial cluster are entered in the bad block file so they cannot be used.

The bad block entries are stored in descending LBN order. The bad block table is searched until an entry is found with a starting LBN that is lower than the current entry. The table is then extended, and the current entry is inserted in its proper position.

Contiguous or overlapping areas are also merged. Neighboring entries are compared to see if their LBNs are adjacent; if they are, the two entries are combined to form one entry expressing a range of LBNs. For more information on the software bad block format, see Section 2.5.3.2.

If the user has specified the /BADBLOCKS qualifier on the INITIALIZE command line (if the user has previously run the Bad Block Locator Utility (BAD), for example), the routine GET_USERBAD is called. It processes any bad block entries the user entered on the command line and puts them in the bad block table. If the entry was specified in cylinder/sector/track format, it is first converted to an LBN.

3.2.4 Performing a Data Security Erase

If the user specified the /ERASE qualifier on the INITIALIZE command line, a data security erase (DSE) is performed on the disk. A DSE can either be:

- A one-step procedure that zeroes the designated blocks on the disk
- An iterative procedure that writes a special pattern to the disk

Both procedures use the $ERAPAT system service, an erase pattern generator. The $ERAPAT code is loadable and may vary from site to site, but the default pattern is 0.

If the default $ERAPAT code is used (or already loaded), the one-step DSE procedure is performed, and the designated blocks are zeroed.

However, if the flag SGN$V_LOADERAPAT[1] in the cell SGN$GL_LOADFLAGS is set, an alternate $ERAPAT has been specified. In this case, the iterative DSE procedure is performed, and it is repeated until $ERAPAT returns a status of SS$_NOTRAN.

Before the disk is erased, the area to be erased is determined, making sure that any bad block data, particularly the factory bad block file, is not overwritten during the procedure. The disk may be erased either from the beginning to the start of the allocated extent or, if there is no bad block data, to the end of the volume.

[1] This flag corresponds to the system parameter LOADERAPAT. This parameter is dynamic (it may be changed on a running system). However, the site-specific $ERAPAT (ERAPAT.EXE) may not be loaded until the system is rebooted.

A DSE is performed only when the user explicitly requests it. The /ERASE qualifier also sets the ERASE volume attribute (HM2$V_ERASE), so when a file on the disk is deleted, it is overwritten with an erase pattern. This can also be accomplished with the command SET VOLUME/ERASE_ON_DELETE.

3.2.5 Locating the Volume Structures

After the data security erase is completed, the allocation routine is called. This routine determines the size and location of each component of the volume's file structure.

An allocation table holds descriptors for all the structure components. The bad block list is part of this table.

Each time a part of the disk is allocated to some structure, an entry is written to this table. The extent of each entry is rounded up or down to the next cluster boundary.

The allocation table index entry for the boot block is inserted. This entry has the effect of locating the boot block at the first available cluster (usually 0).

Next, the primary and secondary home blocks are allocated. If the boot block starts at LBN 0 and the cluster factor is greater than 1, a dummy primary home block cluster is allocated because the real home block starts at LBN 1. If the boot block does not start at LBN 0, the home block search sequence is calculated, and the home block is allocated to the first available block in the sequence. For more information on the home block search sequence, refer to Section 2.5.1.2.

The master file directory, the storage bitmap file, the initial index file, and the alternate index file header are all inserted, in that order, into the allocation table, provided that the /INDEX=END qualifier was not specified on the INITIALIZE command line. This logic results in the best placement of the most frequently referenced portions of the file structure. If the index file is placed at the end of the volume, these files are allocated in reverse order to achieve the same effect.

3.2.6 Building the Storage Bitmap File

After the volume structures have been located in the allocation table, the storage bitmap file (file ID 2,2 or BITMAP.SYS;1) is built and initialized. First, the fields of the storage control block are filled in and written to disk. Then the contents of the bitmap itself are written, making sure that the areas listed in the allocation table (the reserved files) are marked as being in use. The table entries are processed in LBN order, starting with the lowest LBN, to prevent disk thrashing.

The internal allocation table is adjusted to reflect the cluster size. There is one bit in the bitmap for each cluster on the disk. Therefore, each block (512 bytes) in the storage bitmap or allocation table can represent 512 * 8 or 4096 clusters.

3.2.7 Setting Up the Index File

After the SCB and the storage bitmap file have been written to disk, the index file (file ID 1,1 or INDEXF.SYS;1), the boot block, multiple copies of the home block, the index file bitmap, and the initial file headers are all initialized.

First, the boot block is written. Then the current system date and time are obtained with the $GETTIM system service, and the fields of the home block are constructed in order. The quadword containing the date and time is copied to the HM2$W_CREDATE field.

The home block is then written to disk multiple times, depending on the cluster factor. It is written to the remainder of the boot block cluster as well as to the two home block clusters.

The initial index file bitmap is the next data to be written. Its size is sufficient to accommodate the specified maximum number of files on the volume. The first block indicates the reserved files in use. The rest of the blocks contain all 0s.

The first file header to be written is the core image file header. The core image file is used because it is the first file to be written on the volume. The directory back link field points to the MFD. This initial header is then used as the template for the remainder of the reserved files. Essentially, the only information that differs among the file headers is the file ID, the file name, the record attributes (including the record size, the maximum record size, the highest allocated VBN, and the end-of-file block), map pointers (if any), and the checksum.

The first header to be constructed and written using the fields of the core image file header as a guide is the continuation file header. Then the backup journal file header and the pending bad block log file header are constructed and written to disk.

The index file header is the next header to be constructed. Retrieval pointers for the components of the index file are appended to the map area.

The bad block file header and then the storage bitmap file header are constructed. A retrieval pointer is appended to the map area.

The last header to be constructed is the master file directory header. The FH2$V_DIRECTORY bit is set, indicating that this reserved file is a directory; the FAT$M_NOSPAN bit is also set, indicating that the records cannot cross block boundaries. A retrieval pointer is appended to the map area.

3.2.8 Writing the Master File Directory

Last, the contents of the master file directory (file ID 4,4 or 000000.DIR;1) are initialized. In other words, the records for all the reserved files, including the MFD itself, are written into the master file directory.

The MFD records are copied into a zero-filled buffer and written in this order:

- Master file directory record itself
- Backup journal file
- Bad block file
- Pending bad block log file
- Storage bitmap file
- Standard continuation file
- Core image file
- Index file
- Volume set list file

Each record is 24 bytes long (the record byte count field contains 22, which does not include its own 2 bytes). The following chart shows the contents of each directory record in the MFD:

Field Name	Meaning	Value
DIR$W_VERLIMIT	Version limit	1
DIR$B_FLAGS	Flags field	0
DIR$B_NAMECOUNT	File name length	10
DIR$T_NAME	Name string	Name of the file
DIR$C_FID	File ID	File identification number of the file

After the MFD is written to the disk, initialization is complete, and the volume is ready to be mounted.

3.3 Mounting a Volume

Before files or data on a volume can be processed, the volume must be mounted. The Mount Utility (MOUNT) is used to make a disk volume and the files it contains accessible to the file system. It also establishes the necessary resident I/O database structures.

3.3.1 I/O Database

The I/O database is a collection of control blocks generally allocated from nonpaged system (or dynamic) memory. It consists of two types of data structures:

- Those that provide information used by device-oriented components such as drivers, channel control routines, and device interrupt dispatchers.

  They include the unit control block (UCB), the device data block (DDB), the channel request block (CRB), the I/O request packet (IRP), the interrupt dispatch block (IDB), the UNIBUS adapter control block (ADP), and the channel control block[1] (CCB).

- Those that provide information used by file-oriented components such as the file system.

  The file-oriented data structures are created dynamically when a volume is mounted on a device and file activity starts. The information is specific to a particular volume and its files and is maintained as long as the volume remains mounted. There are five major data structures in the I/O database:

  - Volume control block—The VCB is a system data structure to describe volumes.
  - ACP queue block—The AQB identifies a file processor. For an ACP, it contains the I/O queue listhead and a pointer to the ACP process. For the XQP, however, it points to the buffer cache.
  - File control block—The FCB is a collection of per-file process-related information (such as the file highwater mark, UIC, and protection).
  - Window control block—The WCB is the means by which a process looks at a file.
  - Relative volume table—The RVT is a system data structure to track volume sets.

  All these structures are protected so that users can neither read nor write to them.

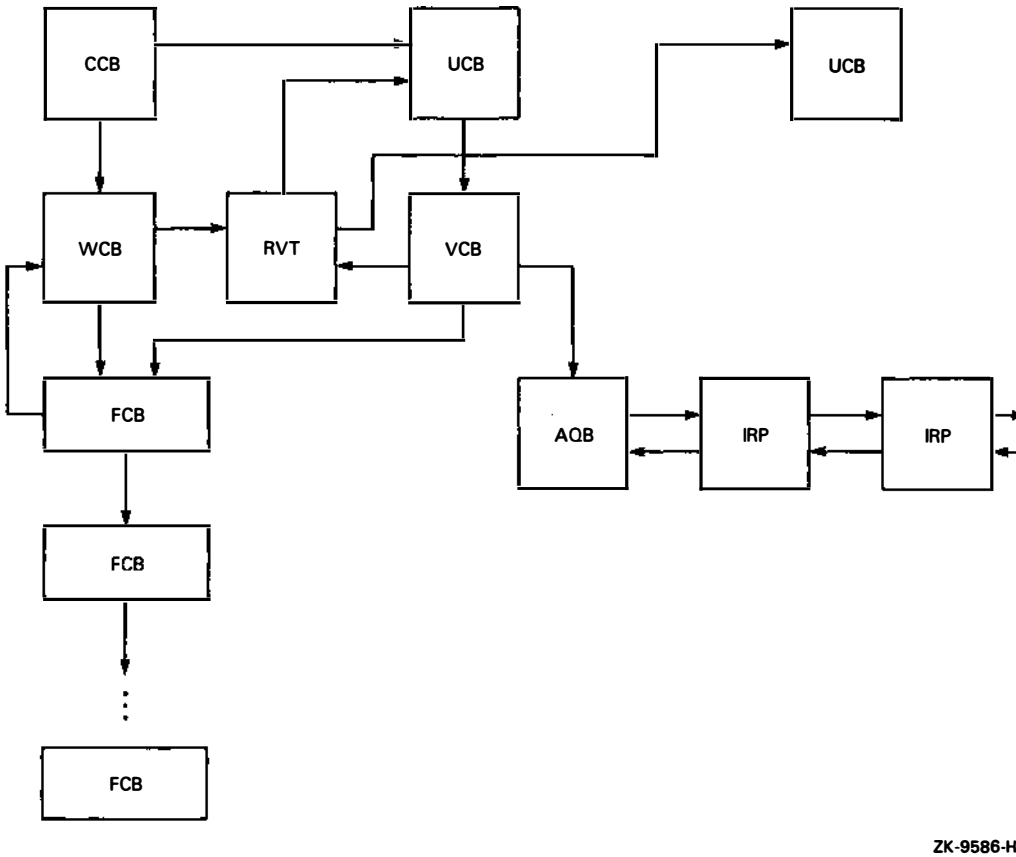
Not only does MOUNT create the data structures of the I/O database and link the structures but it also creates mount-specific data structures such as mounted volume list entries, and logical names associated with the mounted volume, which provide device independence.

The file-oriented data structures are located using a pointer in the appropriate unit control block to find the associated VCB of the mounted volume. Similarly, a pointer in the channel control block is used to locate the WCB for an accessed file.

[1] Channel control blocks are located in P1 space.

Figure 3–1 shows the structures of the I/O database and how they are linked.

Figure 3–1: Relationship Among the File-Oriented Data Structures



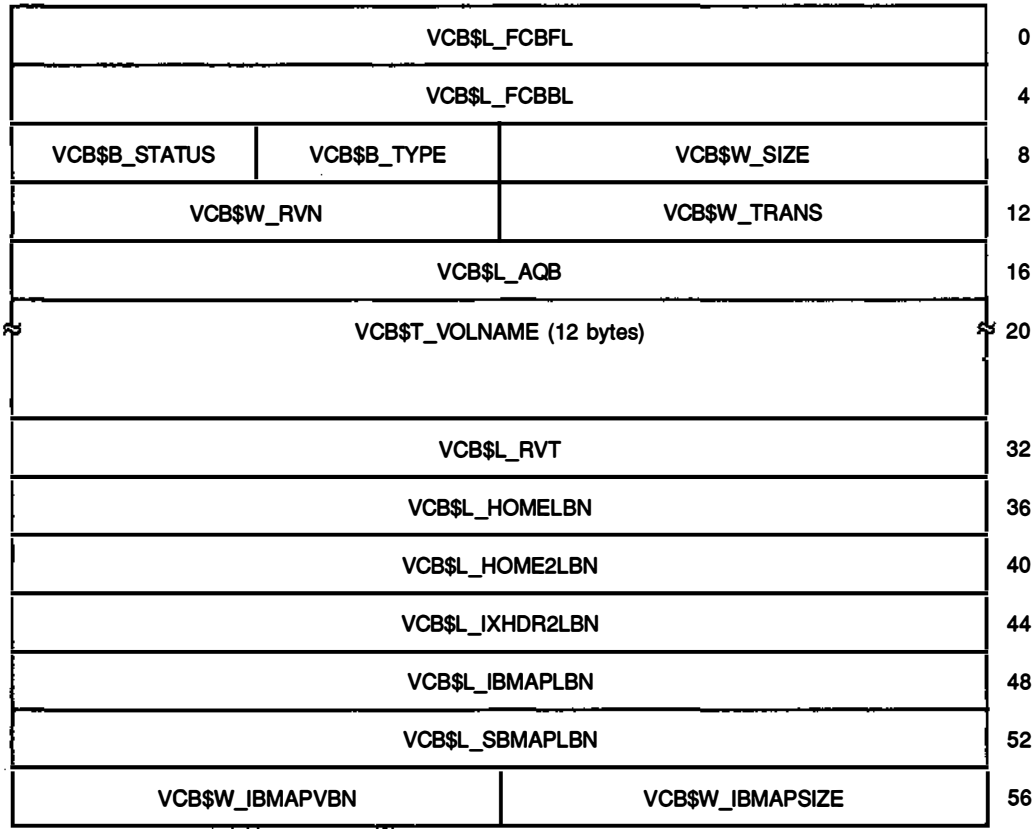
3.3.1.1 Volume Control Block

The **volume control block** (VCB) contains the information needed to control access to a volume. It is created when the volume is mounted, and there is one VCB for each mounted volume (or volume set) on the system. The VCB is permanent for the life of the volume.

It is located by the address in the UCB$L_VCB field in the unit control block. The VCB also contains the addresses for other structures in the I/O database, including the relative volume table and the ACP queue block.

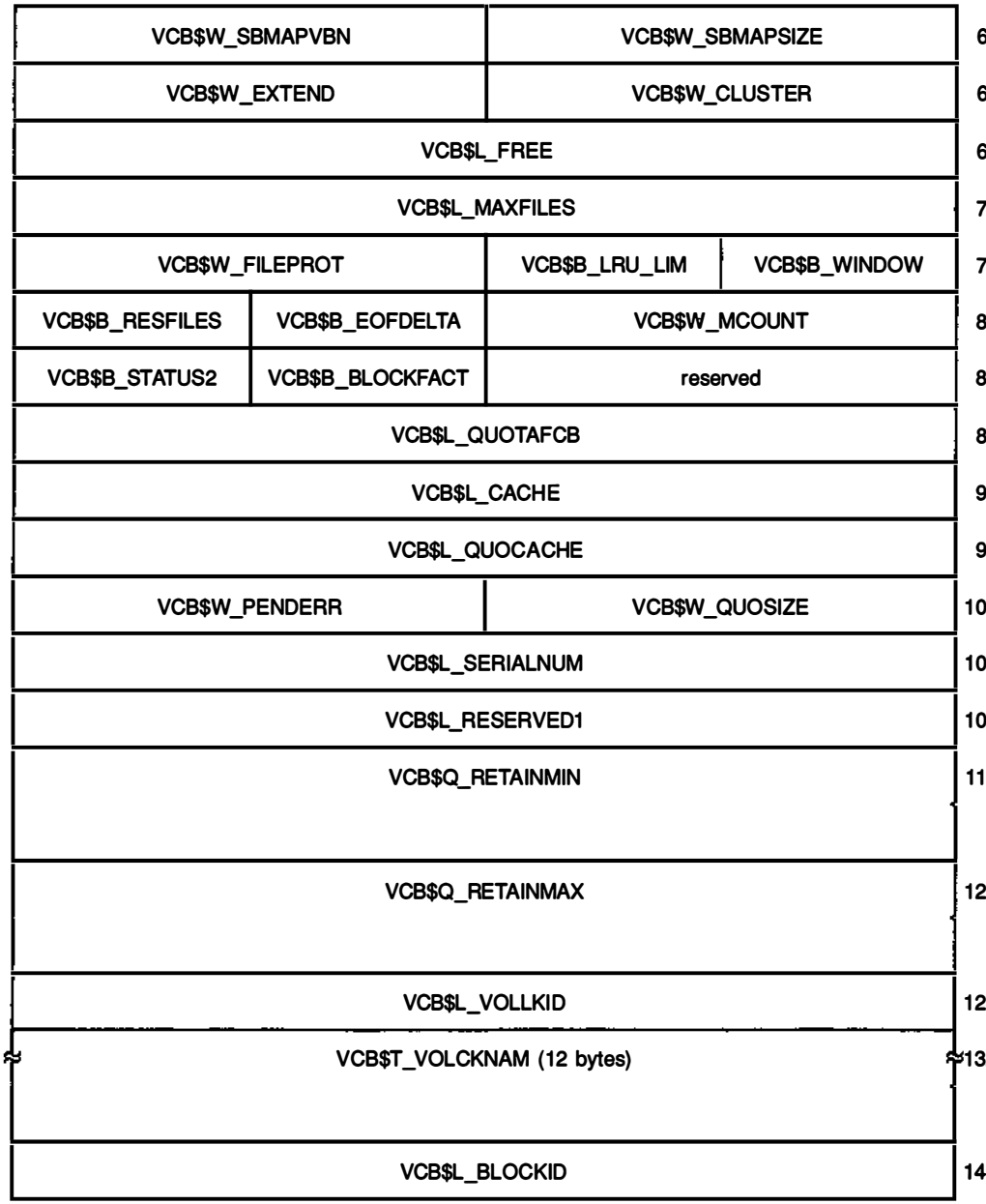
The fields of the VCB are shown in Figure 3–2 and are described in Table 3–1.

Figure 3–2: Format of the Volume Control Block



(continued on next page)

Figure 3–2 (Cont.): Format of the Volume Control Block



(continued on next page)

Figure 3–2 (Cont.): Format of the Volume Control Block

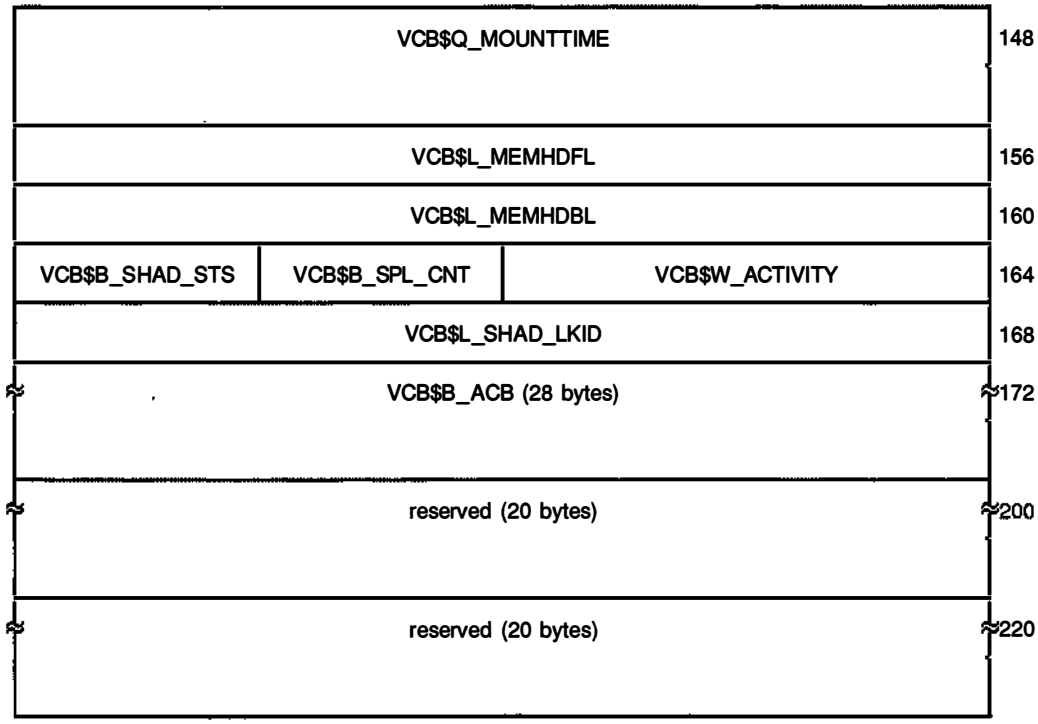


Table 3–1: Contents of the Volume Control Block

Field Name	Description
VCB$L_FCBFL	Forward link of the FCB listhead. All open files on a volume are represented by an FCB linked into this list.
VCB$L_FCBBL	Backward link of the FCB listhead.
VCB$W_SIZE	Size of VCB in bytes.
VCB$B_TYPE	Structure type. This field contains the DYN$C_VCB type code to identify the data structure as a volume control block.
VCB$B_STATUS	Volume status flags. The following flag bits are defined within VCB$B_STATUS:

(continued on next page)

Table 3–1 (Cont.): Contents of the Volume Control Block

Field Name	Description	
	VCB$V_WRITE_IF	Index file is open for write access. This is bit 24.
	VCB$V_WRITE_SM	Storage map is open for write access. This is bit 25.
	VCB$V_HOMBLKBAD	Primary home block is bad. This is bit 26.
	VCB$V_IDXHDRBAD	Primary index file header is bad. This is bit 27.
	VCB$V_NOALLOC	Allocation and deallocation are inhibited because of invalid bitmaps. This is bit 28.
	VCB$V_EXTFID	This bit is no longer used.
	VCB$V_GROUP	Volume is mounted /GROUP. This is bit 30.
	VCB$V_SYSTEM	Volume is mounted /SYSTEM. This is bit 31.
VCB$W_TRANS	Volume transaction count. This field maintains the number of files open on the volume plus the number of I/O request packets in the ACP queue.	
VCB$W_RVN	Relative volume number (RVN). This field contains the RVN of the volume within a multivolume set.	
VCB$L_AQB	Address of ACP queue block.	
VCB$T_VOLNAME	Volume label.	
VCB$L_RVT	Address of the RVT or the UCB. This field may contain either the address of the relative volume table if the volume is part of a volume set or the unit control block for the volume if it is not.	
VCB$L_HOMELBN	LBN of the home block for the volume.	
VCB$L_HOME2LBN	LBN of the alternate home block for the volume.	
VCB$L_IXHDR2LBN	LBN of the alternate index file header.	
VCB$L_IBMAPLBN	LBN of the index file bitmap.	
VCB$L_SBMAPLBN	LBN of the storage bitmap.	
VCB$W_IBMAPSIZE	Size of the index file bitmap in blocks.	

(continued on next page)

Table 3–1 (Cont.): Contents of the Volume Control Block

Field Name	Description
VCB$W_IBMAPVBN	Current VBN in the index file bitmap. This field contains the virtual block number of the block at which to start the next file creation scan.
VCB$W_SBMAPSIZE	Size of the storage bitmap in blocks.
VCB$W_SBMAPVBN	Current VBN in the storage bitmap. This field contains the virtual block number of the block at which to start the next allocation scan.
VCB$W_CLUSTER	Volume cluster size.
VCB$W_EXTEND	Default file extension length for the volume.
VCB$L_FREE	Number of free blocks on the volume.
VCB$L_MAXFILES	Maximum number of files allowed on the volume.
VCB$B_WINDOW	Default window size for the volume.
VCB$B_LRU_LIM	Directory least recently used (LRU) cache entry limit for the volume. This field is not supported.
VCB$W_FILEPROT	Volume default file protection.
VCB$W_MCOUNT	Mount count. This field contains the number of processes that have the volume mounted. This field applies only to shareable mounts.
VCB$B_EOFDELTA	Index file EOF update count. This field is not supported.
VCB$B_RESFILES	Number of reserved files on the volume.
VCB$B_BLOCKFACT	Volume blocking factor.
VCB$B_STATUS2	Second status byte. The following flag bits are defined within VCB$B_STATUS2: VCB$V_WRITETHRU — Write-through caching is enabled for the volume. This is bit 24. VCB$V_NOCACHE — All caching is disabled on volume. This is bit 25. VCB$V_MOUNTVER — Volume can undergo mount verification. This is bit 26. VCB$V_ERASE — Data is erased when blocks are deleted from the file. This is bit 27.

(continued on next page)

Table 3–1 (Cont.): Contents of the Volume Control Block

Field Name	Description	
	VCB$V_NOHIGHWATER	Highwater marking is disabled. This is bit 28.
	VCB$V_NOSHARE	Nonshared mount. This bit starts at bit 29.
	VCB$V_CLUSLOCK	Clusterwide locking is necessary. This is bit 30.
VCB$L_QUOTAFCB	Address of the FCB of the disk quota file.	
VCB$L_CACHE	Address of the volume cache block.	
VCB$L_QUOCACHE	Address of the volume quota cache.	
VCB$W_QUOSIZE	Size of the quota cache in bytes.	
VCB$W_PENDERR	Count of pending write errors.	
VCB$L_SERIALNUM	Volume serial number.	
VCB$L_RESERVED1	This field is reserved.	
VCB$Q_RETAINMIN	Minimum file retention period in ADT format.	
VCB$Q_RETAINMAX	Maximum file retention period in ADT format.	
VCB$L_VOLLKID	Volume lock ID.	
VCB$T_VOLCKNAM	Name for volume lock.	
VCB$L_BLOCKID	Volume blocking lock. This field contains the 12-byte unique volume identifier. It is used, along with the VCB$W_ACTIVITY field, to stall activity on a single volume. It is also called an **activity blocking lock**, or **blocking lock**. See Section 8.3.5 for more information on the blocking lock. However, if the volume is a volume set member, the corresponding fields in the RVT are used instead, so blocking occurs over the entire volume set. See Section 3.3.1.5 for more information.	
VCB$Q_MOUNTTIME	Volume mount time.	
VCB$L_MEMHDFL	Controller shadow set members queue header forward link. This field contains the linked list pointer from the virtual unit VCB through the list of physical member VCBs.	
VCB$L_MEMHDBL	Controller shadow set members queue header backward link. This field contains the linked list pointer from the virtual unit VCB through the list of physical member VCBs.	

(continued on next page)

Table 3–1 (Cont.): Contents of the Volume Control Block

Field Name	Description
VCB$W_ACTIVITY	Activity count flag. This field determines whether or not processing can be performed on the volume. If the low bit of this field is set (that is, it contains an odd value), status is normal, and volume activity can proceed. If the field contains a value of 0, the volume is idle, and further activity is blocked. If the field contains an even, nonzero value, the volume is not idle, and further activity is blocked.
VCB$B_SPL_CNT	Count field of devices spooled to the volume. A volume that has devices spooled to it cannot be dismounted, so DISMOUNT checks this field for a value of 0. This field is incremented by the command SET DEVICE/SPOOLED.
VCB$B_SHAD_STS	Controller shadowing rebuild state flags.
VCB$L_SHAD_LKID	Controller shadowing rebuild synchronization lock ID. Controller shadowing uses this lock through an AST to the swapper to synchronize a shadow set rebuild operation between different VAXcluster nodes.
VCB$B_ACB	AST control block for a blocking AST.

3.3.1.2 Window Control Block

The **window control block** (WCB) has two main purposes:

- It provides storage for access control information.
- It contains a set of mapping pointers that allow the virtual block numbers of a file to be mapped to the logical block numbers on a disk.

In other words, the WCB contains the information necessary to transfer information from the disk.

The WCB is located in nonpaged pool. It is pointed to by the CCB$L_WCB field in the channel control block, and it points to both the relative volume table and the file control block associated with the file.

A WCB is local to one access, and when another access to a file is sought, another WCB is built. Multiple WCBs may be associated with a single file. Of all the data structures in the I/O database, it is the most likely to be adjusted because of window turns. Also, accessing and deaccessing a file causes the WCB to be discarded.

The fields of the WCB are shown in Figure 3–3 and are described in Table 3–2.

Figure 3–3: Format of the Window Control Block

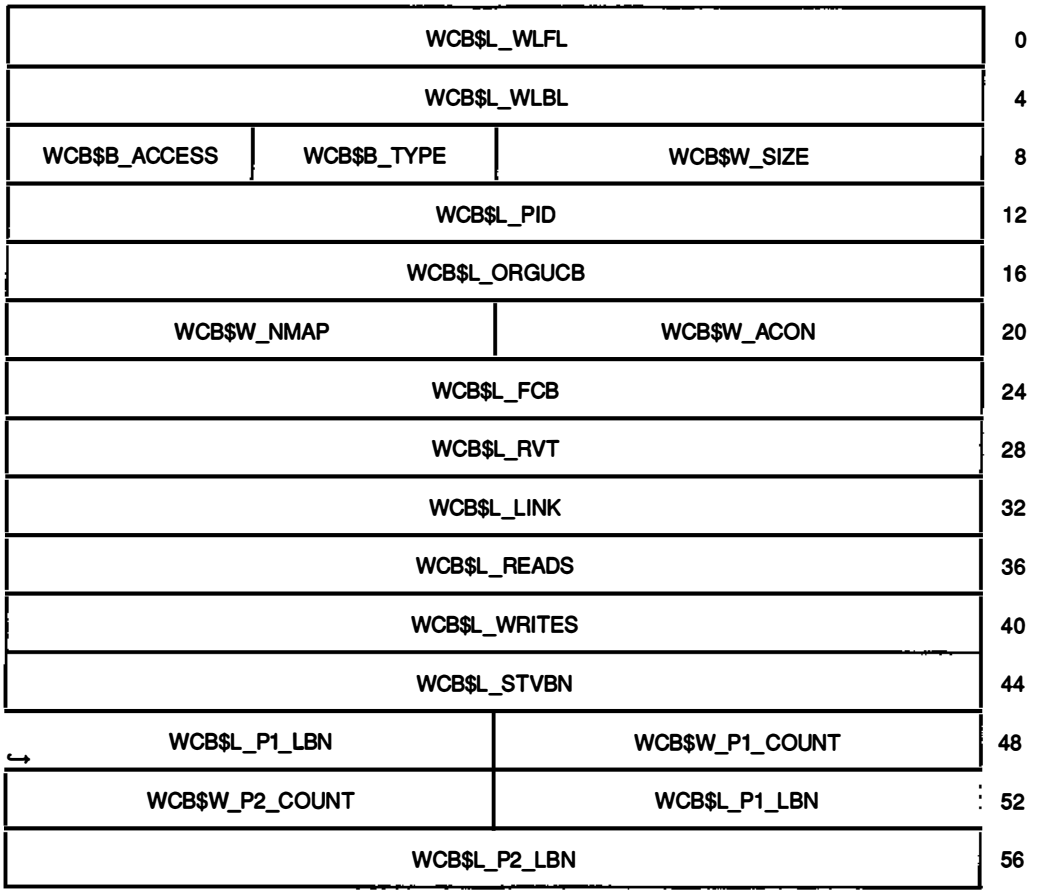


Table 3–2: Contents of the Window Control Block

Field Name	Description	
WCB$L_WLFL	Window list forward link. This field contains the forward link of the window list that connects all windows for a given file to their respective file control blocks.	
WCB$L_WLBL	Window list backward link.	
WCB$W_SIZE	Size of window block in bytes.	
WCB$B_TYPE	Structure type. This field contains the DYN$C_WCB type code to identify the data structure as a window control block.	
WCB$B_ACCESS	Access control byte. The following flag bits are defined within WCB$B_ACCESS:	
	WCB$V_READ	Set if read access is allowed. This is bit 24.
	WCB$V_WRITE	Set if write access is allowed. This is bit 25.
	WCB$V_NOTFCP	Set if the file is not accessed by the standard file system. This is bit 26.
	WCB$V_SHRWCB	Shared window. This is bit 27.
	WCB$V_OVERDRAWN	File accessor has overdrawn the quota allotted to accessor's process. This is bit 28.
	WCB$V_COMPLETE	Set if the window maps the entire file. This is bit 29.
	WCB$V_CATHEDRAL	Large, complex window used to map the file completely. Contiguous files are always completely mapped, but noncontiguous file may or may not be completely mapped. However, if the file is opened with a cathedral window, complete mapping may be ensured. All the mapping information will be read from the file headers (from the map area or areas) into a WCB, so no window turn is ever necessary. This is bit 30.

(continued on next page)

Table 3–2 (Cont.): Contents of the Window Control Block

Field Name	Description	
	WCB$V_EXPIRE	File expiration date may need to be set. This is bit 31.
WCB$L_PID	Process ID of the accessor process.	
WCB$L_ORGUCB	Address of the original UCB from the CCB.	
WCB$W_ACON	Access control information. Note that these bits track the bits in the FIB$L_ACCTL field. The following flag bits are defined within WCB$W_ACON:	
	WCB$V_NOWRITE	Other writers are not allowed. This is bit 0.
	WCB$V_DLOCK	Deaccess locking is enabled. This is bit 1.
	WCB$V_SPOOL	File is spooled when it is closed. This is bit 4.
	WCB$V_WRITECK	Write checking is enabled. This is bit 5.
	WCB$V_SEQONLY	Sequential access only is permitted. This is bit 6.
	WCB$V_WRITEAC	Write access is permitted. This is bit 8.
	WCB$V_READCK	Read checking is enabled. This is bit 9.
	WCB$V_NOREAD	Other readers are not allowed. This is bit 10.
	WCB$V_NOTRUNC	Truncation is not allowed. This is bit 11.
	The following flag bits defined within WCB$W_ACON do not track the bits in the FIB$L_ACCTL field:	

(continued on next page)

Table 3–2 (Cont.): Contents of the Window Control Block

Field Name	Description	
	WCB$V_NOACCLOCK	Arbitration lock checking is not performed (that is, the file was opened with the FIB$V_NOLOCK flag set). This is bit 2.
	WCB$V_WRITE_TURN	Window turns are forced during write operations. This bit starts at bit 12.
WCB$W_NMAP	Number of mapping pointers.	
WCB$L_FCB	Address of the FCB.	
WCB$L_RVT	Address of either the RVT or the UCB (if the volume is not a member of a volume set).	
WCB$L_LINK	Link to the next window segment.	
WCB$L_READS	Count of read operations performed.	
WCB$L_WRITES	Count of write operations performed.	
WCB$L_STVBN	Starting VBN mapped by the window.	
WCB$W_P1_COUNT	Count field of the first pointer in the WCB.	
WCB$L_P1_LBN	Disk address (LBN field) of the first pointer.	
WCB$W_P2_COUNT	Count field of the second pointer.	
WCB$L_P2_LBN	Disk address (LBN field) of the second pointer. Format of retrieval pointer.	

3.3.1.3 ACP Queue Block

The AQB represents an instance of an ACP or XQP. For an ACP, the AQB contains the listhead of the queue of I/O request packets. On the other hand, the XQP uses the listhead for other purposes, and keeps its queue listhead in the P1 space of each process.

The AQB is the communication path between the Queue I/O Request ($QIO) system service and the XQP. Each process has its own AQB, while one AQB represents the XQP in all processes.

The fields of the AQB are shown in Figure 3–4 and are described in Table 3–3.

Figure 3–4: Format of the ACP Queue Block

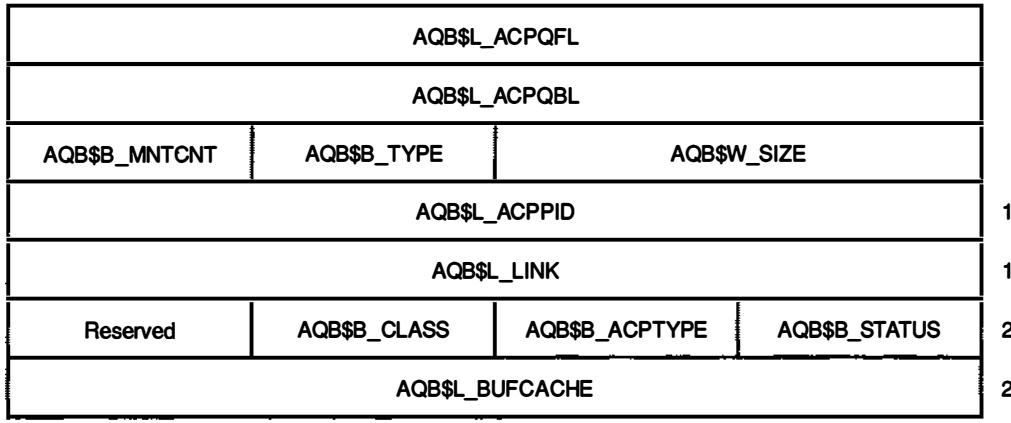


Table 3–3: Contents of the ACP Queue Block

Field Name	Description
AQB$L_ACPQFL	ACP IRP queue listhead forward link. This field points to the first IRP in the queue. However, if the ACB represents an XQP, this queue listhead is used to synchronize access to the buffer cache.
AQB$L_ACPQBL	ACP IRP queue listhead backward link. This field points to the last IRP in the queue.
AQB$W_SIZE	Size of the AQB in bytes.
AQB$B_TYPE	Structure type. This field contains the DYN$C_AQB type code to identify the data structure as an ACP queue block.
AQB$B_MNTCNT	ACP mount count. This field contains the number of volumes being serviced by an ACP process.
AQB$L_ACPPID	Process identification. This field contains the process identication (PID) of the ACP process servicing the queue. If the ACP represents an XQP, this field contains 0.
AQB$L_LINK	AQB list linkage.

(continued on next page)

Table 3–3 (Cont.): Contents of the ACP Queue Block

Field Name	Description	
AQB$B_STATUS	Status byte. The following flag bits are defined within AQB$B_STATUS:	
	AQB$V_UNIQUE	ACP is unique to this device. This is bit 0.
	AQB$V_DEFCLASS	ACP is default for this class. This is bit 1.
	AQB$V_DEFSYS	ACP is default for the system. This is bit 2.
	AQB$V_CREATING	ACP is currently being created. This is bit 3.
	AQB$V_XQIOPROC	Extended QIO processor is being used. This is bit 4.
AQB$B_ACPTYPE	ACP type code—magnetic tape ACP (MTAACP), remote ACP (REMACP), or network ACP (NETACP).	
AQB$B_CLASS	ACP class code.	
AQB$L_BUFCACHE	Pointer to the buffer cache if this is an XQP.	

3.3.1.4 File Control Block

The **file control block** (FCB) contains the information needed to control access to a file. It is created when a file is accessed for the first time; subsequent accesses to this file must use the same FCB.

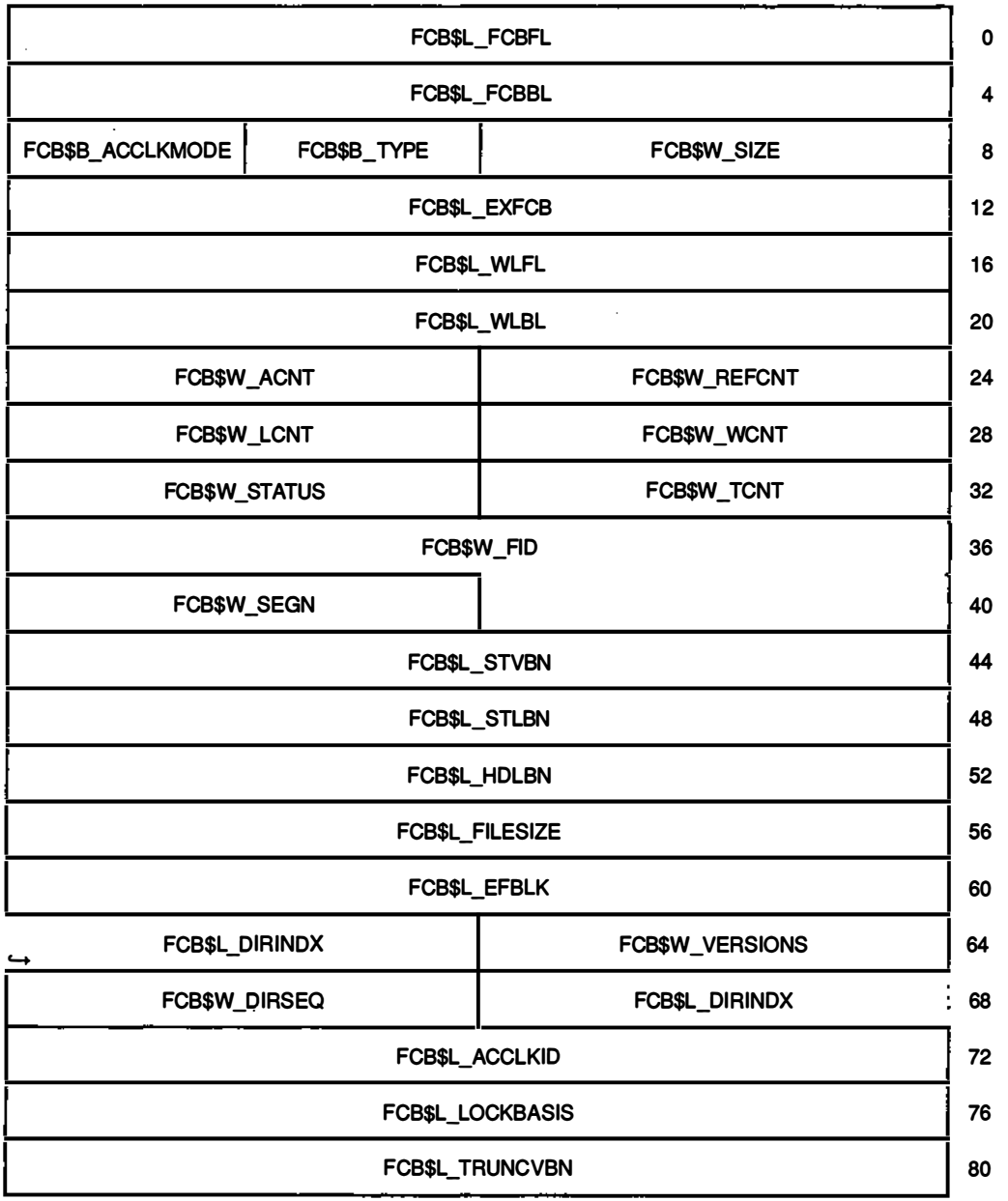
FCBs for recently used directories are retained to optimize repeated access to directories.

The FCB points to the file header for additional mapping information about the file. It is pointed to by the WCB$L_FCB field. FCBs are chained together in a doubly linked list.

A file is represented by one FCB per node, but copies of that FCB exist on every node of a VAXcluster. In this way, an FCB may be considered global to a VAXcluster but local to a system.

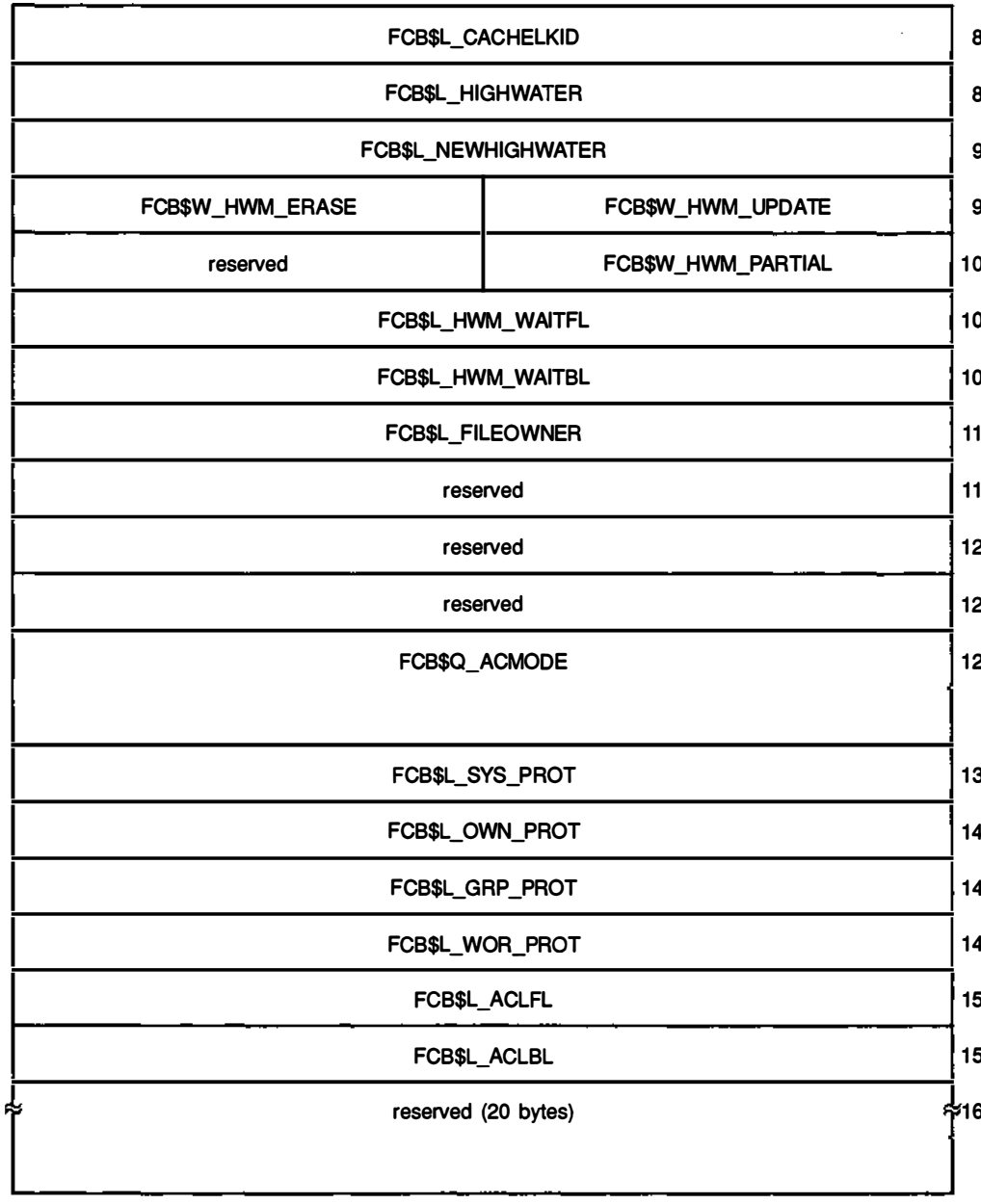
The fields of the FCB are shown in Figure 3–5 and are described in Table 3–4.

Figure 3–5: Format of the File Control Block



(continued on next page)

Figure 3–5 (Cont.): Format of the File Control Block



(continued on next page)

Figure 3–5 (Cont.): Format of the File Control Block

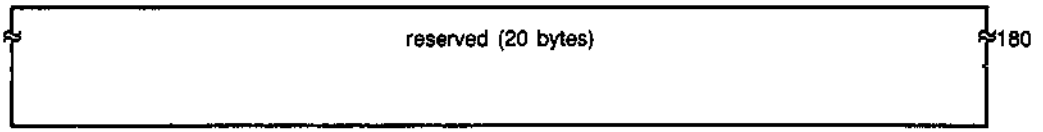


Table 3–4: Contents of the File Control Block

Field Name	Description
FCB$L_FCBFL	FCB list forward link. This field is the forward link for linking the FCB to the chain of FCBs off the volume control block.
FCB$L_FCBBL	FCB list backward link. This field is the backward link for linking the FCB to the chain of FCBs off the volume control block.
FCB$W_SIZE	Size of FCB in bytes.
FCB$B_TYPE	Structure type. This field contains the DYN$C_FCB type code to identify the data structure as a file control block.
FCB$B_ACCLKMODE	Arbitration lock mode. This field contains the highest lock mode of an accessor on this node of the VAXcluster.
FCB$L_EXFCB	Address of the extension FCB. This field contains the address of the extension file control block. If one does not exist, this field contains a 0. An extension FCB is created for each extension header of a multiheader file.
FCB$L_WLFL	Forward link of the window listhead.
FCB$L_WLBL	Backward link of the window listhead.
FCB$W_REFCNT	Reference count. This field gives the total references to this FCB, which represents the number of channels active to the file. In other words, it indicates the number of processes that are currently accessing the file.
FCB$W_ACNT	File access count. The field gives the number of users that currently have the file open for access. This count does not include accesses with the FIB$V_NOLOCK flag, so it can be less than the reference count.
FCB$W_WCNT	File writer count. This field gives the number of channels open to the file that have the ability to write to the file.

(continued on next page)

Table 3–4 (Cont.): Contents of the File Control Block

Field Name	Description	
FCB$W_LCNT	File lock count. This field gives the number of accessors that have the file locked against writers.	
FCB$W_TCNT	Count of truncate locks. This field gives the number of accessors that have the file locked to prevent truncation.	
FCB$W_STATUS	File status. The following flag bits are defined within FCB$W_STATUS:	
	FCB$V_DIR	FCB is a directory LRU entry. This is bit 16.
	FCB$V_MARKDEL	File is marked for deletion. This is bit 17.
	FCB$V_BADBLK	Bad block encountered in file. This is bit 18.
	FCB$V_EXCL	File is exclusively accessed. This is bit 19.
	FCB$V_SPOOL	File is an intermediate spool file. This is bit 20.
	FCB$V_RMSLOCK	File is open with RMS record locking. This is bit 21.
	FCB$V_ERASE	Data will be erased when the blocks are removed from the file. This is bit 22.
	FCB$V_BADACL	ACL is corrupt. This is bit 23.
	FCB$V_STALE	FCB must be reconstructed from the file header. This is bit 24.
	FCB$V_DELAYTRNC	Delayed truncation is pending against the file. This is bit 25.
FCB$W_FID	File identifier.	
FCB$W_SEGN	File segment number.	
FCB$L_STVBN	Starting VBN. This field contains the starting virtual block number of the file section represented by this FCB.	
FCB$L_STLBN	Starting LBN. This field contains the starting logical block number of the file, if it is contiguous. If the file is not contiguous, this field contains a value of 0.	
FCB$L_HDLBN	LBN of the file header.	
FCB$L_FILESIZE	File size in blocks.	

(continued on next page)

Table 3–4 (Cont.): Contents of the File Control Block

Field Name	Description
FCB$L_EFBLK	End-of-file VBN.
FCB$W_VERSIONS	Maximum number of versions in directory. This field applies to directory files only.
FCB$L_DIRINDX	Directory index block pointer. This field contains a pointer to the directory index block in the buffer cache that corresponds to this directory file. It is used only for directory FCBs.
FCB$W_DIRSEQ	Directory use sequence number. This field applies to directory files only.
FCB$L_ACCLKID	Access lock ID.
FCB$L_LOCKBASIS	Lock basis for this FCB. This field contains the basis (a file number and an RVN) for building the resource name for locks taken against this file.
FCB$L_TRUNCVBN	VBN for delayed truncation.
FCB$L_CACHELKID	Cache interlock lock ID.
FCB$L_HIGHWATER	Highwater mark in file.
FCB$L_NEWHIGHWATER	Highwater mark of a pending write operation. This field contains the highest block in the file that is in the process of being written.
FCB$W_HWM_UPDATE	Count of write operations in progress that affect the highwater mark.
FCB$W_HWM_ERASE	Count of highwater mark erase operations in progress.
FCB$W_HWM_PARTIAL	Count of partially validated erase operations. This field contains a count of pending highwater mark erase operations that were truncated by end of file. (Corresponding IRPs are flagged with the IRP$V_PART_HWM bit.)
FCB$L_HWM_WAITFL	Highwater mark update queue forward link. This field contains the queue head for virtual I/Os waiting for conflicting highwater-mark-related operations to complete. Pending write operations are stored on the front of the queue.
FCB$L_HWM_WAITBL	Highwater mark update queue backward link. Pending read operations are stored on the back of the queue.
FCB$L_FILEOWNER	File owner UIC.
FCB$Q_ACMODE	Access mode protection vector.
FCB$L_SYS_PROT	System protection word.
FCB$L_OWN_PROT	Owner protection word.

(continued on next page)

Table 3–4 (Cont.): Contents of the File Control Block

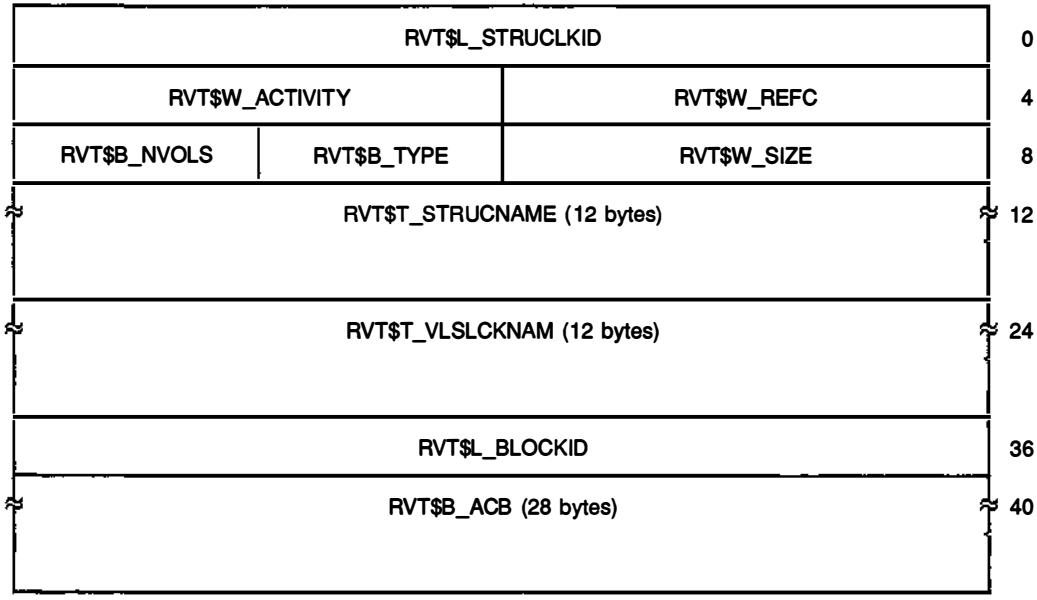
Field Name	Description
FCB$L_GRP_PROT	Group protection word.
FCB$L_WOR_PROT	World protection word.
FCB$L_ACLFL	Access control list forward link.
FCB$L_ACLBL	Access control list backward link.

3.3.1.5 Relative Volume Table

The **relative volume table** (RVT) contains the information to associate the volumes of a multivolume set with the address of the UCB of the unit on which each of the volumes is mounted. In other words, there is one RVT per volume set. It is pointed to by both the window control block and the volume control block. The RVT may point to multiple UCBs.

The fields of the RVT are shown in Figure 3–6 and are described in Table 3–5.

Figure 3–6: Format of the Relative Volume Table



(continued on next page)

Figure 3–6 (Cont.): Format of the Relative Volume Table

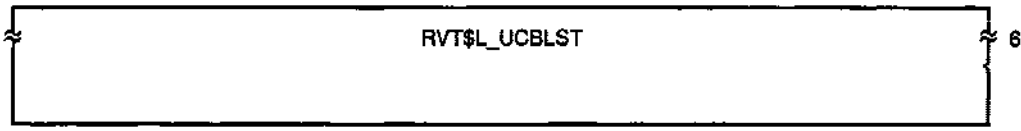


Table 3–5: Contents of the Relative Volume Table

Field Name	Description
RVT$L_STRUCLKID	Lock ID of volume set lock.
RVT$W_REFC	Reference count, which is the number of volumes in the volume set that are currently mounted.
RVT$W_ACTIVITY	Activity count flag. This field determines whether or not processing can be performed on the volume set. If the low bit of this field is set (that is, it contains an odd value), status is normal, and volume set activity can proceed. If the field contains a 0, the volume set is idle, and further activity is blocked. If the field contains an even, nonzero value, the volume is not idle, and further activity is blocked.
RVT$W_SIZE	Size of RVT in bytes.
RVT$B_TYPE	Structure type. This field contains the DYN$C_RVT type code to identify the data structure as a relative volume table.
RVT$B_NVOLS	Number of volumes in the set.
RVT$T_STRUCNAME	Volume set name.
RVT$T_VLSLCKNAM	Volume set lock name.
RVT$L_BLOCKID	Blocking lock ID. This field contains the 12-byte unique volume set identifier. It is used, along with the RVT$W_ACTIVITY field, to stall volume set activity. It is also called an **activity blocking lock**.
RVT$B_ACB	AST control block for blocking AST.

(continued on next page)

Table 3–5 (Cont.): Contents of the Relative Volume Table

Field Name	Description
RVT$L_UCBLST	Addresses of UCBs. This field contains the beginning of a table of UCB addresses for all volumes in the set. For a given relative volume number (RVN), the UCB can be accessed by using the RVN as an index into the RVT$L_UCBLST table. For example, the UCB address of relative volume number 1 is at location RVT$L_UCBLST; the UCB address of relative volume number 2 is at location RVL_UCBLST + 4; and so on.

3.3.2 Processing the Volume Mount

A request to mount a volume occurs because of initial system startup or because a specific request is received from a user or an operator to mount a volume. The Mount Utility (MOUNT), which makes a disk available for processing, is a privileged shareable image. Therefore, it has the privilege to change mode to kernel so that it can allocate and build the resident I/O database components.

However, MOUNT does as much work as possible in executive mode. For example, all the disk blocks (including the index file, the storage bitmap headers, and the storage bitmap itself) are read in executive mode.

The VMOUNT module is the main routine of the $MOUNT system service. It contains the general control flow of the mount operation.

The module MOUNTIMG is located in the VMOUNT image. This image acquires the DCL MOUNT command line from the CLI parser, parses this command line, and calls the $MOUNT system service.

The $MOUNT system service performs the actual mount operation.

3.3.2.1 Obtaining User Input

The main routine SYS$VMOUNT processes the parameters the user entered on the command line. The user's current privilege mask is saved, and the following amplified privileges are granted:

Privilege	Meaning
ACNT	Disable accounting
ALTPRI	Set base priority higher than allotment
BUGCHK	Make bugcheck error log entries

Privilege	Meaning
BYPASS	Disregard protection
DETACH	Create detached processes of arbitrary UIC
EXQUOTA	Exceed disk quota
GROUP	Control processes in the same group
MOUNT	Execute mount volume QIO
PHY_IO	Issue physical I/O requests
PSWAPM	Change process swap mode
TMPMBX	Temporary mailbox
SETPRV	Enable any privilege bit
SYSLCK	Lock systemwide resources
WORLD	Control any process

SYS$VMOUNT transfers control to the routine VMOUNT_ENVELOPE, which serves as the base call frame for all the executive mode code, and it intercepts all executive mode conditions. It calls the MOUNT_VOLUME routine, which attempts to mount the volume.

If the user specified the /SHARE, /GROUP, or the /SYSTEM qualifier on the command line, the I/O database must be searched for a matching volume label.

The volume lock, which correctly serializes simultaneous shared mounts, is released by the SYS$VMOUNT routine when the volume has been completely and successfully mounted.

3.3.2.2 Searching for a Mountable Device

A volume may be mounted in one of two ways:

- Shared mount—Specified by the /SHARE, /GROUP, /SYSTEM, or /CLUSTER qualifier. The device on which the volume is mounted is not allocated; the volume may be accessed from more than one process.
- Private mount—Default. The device that the volume is mounted on is allocated to the job from which the mount request was issued.

If the volume is to be mounted with the /SHARE, /GROUP, or /SYSTEM qualifier, the I/O database is searched for a device with the label specified on the command line. The database must be locked during the search. The list of device data blocks (DDBs) is walked, and the UCB list off each DDB is followed to find file-structured devices that are mounted but not allocated. If the search is successful, the UCB address is returned.

If the volume is being mounted with the /SYSTEM or the /GROUP qualifier, an error is signaled if a duplicate volume name is found. Only if the volume is being mounted for sharing can a duplicate volume name be tolerated. In this case, the mount count is incremented, and the volume found is successfully mounted.

If the volume is not found or it is being mounted with the /NOSHARE qualifier, the I/O database is searched again for a device that can be mounted. If a private mount is requested, the IOC$V_ALLOC flag is set to take out an exclusive lock. When the device is found, a lock is taken out against the allocation class device name. An exclusive mode lock is acquired if the device is being allocated (that is, the volume is being mounted privately) and a protected write mode lock is acquired if the volume is being mounted publicly. The lock is taken in noqueue mode, so that if the device is in use elsewhere in the cluster, the lock request will fail. Once the device is successfully locked, it is allocated with the following actions:

- The access mode is set to the access mode of the caller.
- The DEV$V_ALL bit is set in the UCB$L_DEVCHAR field, indicating that it is allocated.
- The reference count is incremented in the UCB$W_REFC field.
- The device owner is set.

Once the device is acquired, the **mount lock** or **mount interlock** is taken in exclusive mode to synchronize all mounts on this device. It has the following form:

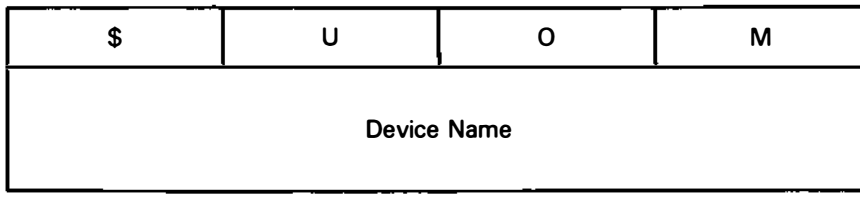
```
MOU$<allocation-class-device-name>
```

This lock is also taken in noqueue mode.

If the MOU$ interlock fails, the device is released, and MOUNT queues for the MOU$ lock. Once the lock is granted, it is released. MOUNT waits a short random interval to prevent "livelock" with other processes, and then repeats the device acquisition procedure.

Figure 3–7 shows the format of the mount lock.

Figure 3–7: Mount Synchronization Lock



ZK-9729-HC

3.3.2.3 Setting Up Device Context

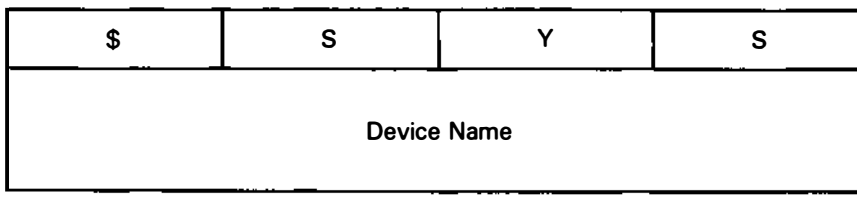
Once the device has been acquired, a channel to the device is obtained with the $GETCHAN system service. The device characteristics are obtained with $GETDVI, and the device type is validated. The mount qualifiers are checked to ensure that they are consistent with the device type.

Next, the device context must be obtained to ensure that mounts of the same device from different nodes in a cluster are consistent. The mount context relevant to the device and volume locks must be initialized by acquiring the value block of the **device lock**, if it exists. The device lock has the following form:

```
SYS$<device-name>
```

Figure 3–8 shows the format of the device lock.

Figure 3–8: Device Allocation Lock



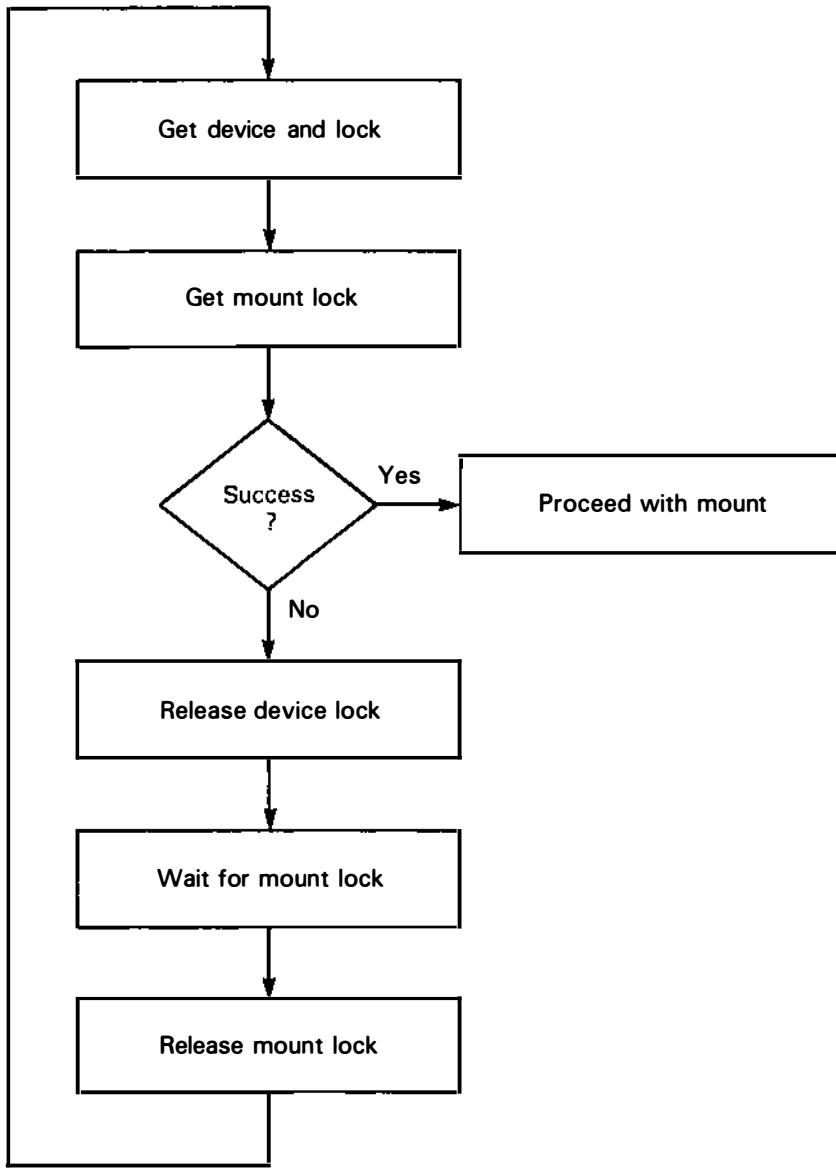
ZK-9728-HC

Lock modes for the device lock have the following meanings:

Lock Mode	Meaning
Concurrent read	Channel assigned or mounted.
Protected write	Mount is in progress.
Exclusive	Device is allocated.

Figure 3–9 shows how the mount and device locks are acquired.

Figure 3–9: Device Synchronization Flow



The mount context for a device (if it is already mounted) is contained in the **device lock value block**. The lock ID is obtained from the UCB$L_LOCKID field. The device lock value block is checked when a volume is mounted shared or clusterwide. The device context of the mounted volume is compared with the qualifiers specified in the MOUNT command to see if the two are compatible. If they are incompatible, an error is returned.

The value block of the device lock is heavily used. For example, it tracks the following MOUNT information:

- Mount mode
- Structure level of the volume
- UIC
- Protection
- Whether the volume is mounted read-only or read/write

This information is all used by MOUNT to guarantee consistency in the way the volume is mounted across the cluster.

Figure 3–10 shows the value block for the device lock.

Figure 3–10: Device Lock Value Block

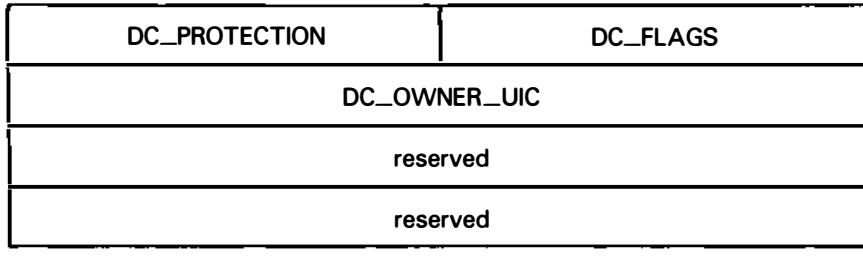


Table 3–6 shows the fields of the device allocation lock value block.

Table 3–6: Contents of the Device Lock Value Block

Field Name	Description	
DC_FLAGS	Device usage flags. This field corresponds to the DAL$W_FLAGS field. The following flag bits are defined within DC_FLAGS:	
	DC_NOTFIRST_MNT	Not the first time mounted. This bit is clear if the volume has not been mounted elsewhere. It corresponds to the DAL$V_NOTFIRST_MNT bitfield mask.
	DC_FOREIGN	The device was mounted with the MOUNT/FOREIGN command. This bit corresponds to the DAL$V_FOREIGN bitfield mask.
	DC_GROUP	The device was mounted with the MOUNT/GROUP command. This bit corresponds to the DAL$V_GROUP bitfield mask.
	DC_SYSTEM	The device was mounted with the MOUNT/SYSTEM command. This bit corresponds to the DAL$V_SYSTEM bitfield mask.
	DC_WRITE	Write access allowed. This bit corresponds to the DAL$V_WRITE bitfield mask.
	DC_NOQUOTA	Quota checking disabled. This bit corresponds to the DAL$V_NOQUOTA bitfield mask.
	DC_OVR_PROT	Override protection. This bit corresponds to the DAL$V_OVR_PROT bitfield mask.
	DC_OVR_OWNUIC	Override volume ownership. This bit corresponds to the DAL$V_OVR_OWNUIC bitfield mask.
	DC_NOINTERLOCK	Access is not VAXcluster interlocked. This bit corresponds to the DAL$V_NOINTERLOCK bitfield mask.
	DC_SHADOW_MBR	Shadow set member. This bit corresponds to the DAL$V_SHADOW_MBR bitfield mask.

(continued on next page)

Table 3–6 (Cont.): Contents of the Device Lock Value Block

Field Name	Description
DC_PROTECTION	Volume protection.
DC_OWNER_UIC	Volume owner UIC.

If the device lock value block is zero, the process is considered the first mounter on this device.

3.3.2.4 Establishing the Volume Defaults

The actual process of mounting an ODS–2 disk is handled by the routine MOUNT_DISK2 in the module MOUDK2. This routine does as much preliminary work as possible in executive mode. All the disk blocks (including the index file, the storage bitmap headers, and the storage bitmap itself) are read in executive mode so that the program can be aborted without corrupting the reserved files or leaving the structures of the I/O database in an undefined state. Also, prototype control blocks are built in local storage and then copied into system pool for the same reason.

The process and volume owner UIC are obtained. The current PCB is located in the scheduler database, and the process UIC is read from the PCB$L_UIC field. The volume owner UIC is read from the HM2$L_VOLOWNER field in the private copy of the home block. Privilege checks are made for overriding volume protection and for options requiring operator privilege.

The volume set name is established, either from the /BIND qualifier on the command line or from the HM2$T_STRUCTNAME field in the home block. If both are present, they must match.

The system defaults are checked to establish the following specialized cache sizes:

- **Extent cache**—If the size of the extent cache has not been set and the user did not explicitly disable caching for the volume, the default extent cache size is established from the value set with the ACP_EXTCACHE system parameter. Otherwise, the cache size is set to 0, disabling extent caching for the volume.

  The default limit of the volume space to which the extent cache can point is established using the ACP_EXTLIMIT system parameter.

- **FID cache**—The default FID cache size is established from the value set with the ACP_FIDCACHE system parameter. If no FID cache is needed on the volume, the cache size is set to 1.

- **Quota cache**—The default quota cache size is established from the value set with the ACP_QUOCACHE system parameter. The cache size is contained in the VCB$W_QUOSIZE field, and the quota file itself is always placed on RVN 1. If no quota cache is needed on the volume (or on the remaining members of a volume set), the cache size is set to 0.

The transaction count and the mount count are both set to 1 in the prototype VCB. Other fields of the prototype VCB are filled in using the fields of the home block, including the fields HM2$V_ERASE, HM2$L_SERIALNUM, and HM2$V_NOHIGHWATER. If the volume is being mounted with the /GROUP qualifier, the VCB$V_GROUP bit is set to 1. If it is being mounted with the /SYSTEM qualifier, the VCB$V_SYSTEM bit is set to 1.

The value of the VCB$L_HOMELBN field of the prototype VCB is copied from the LBN of the primary home block. The value of the VCB$L_HOME2LBN field is likewise copied from the HM2$L_ALHOMELBN field of the home block. If the value of the two fields is equal, then the VCB$V_HOMBLKBAD bit is set, indicating that the primary home block is invalid.

The device blocking factor is obtained from the device information block. The index file bitmap LBN, the volume cluster factor, and the default window size are also filled in using the fields in the home block. If the current number of window pointers is 0, the default number of window pointers is set to 7. However, if the volume is being mounted with the /SYSTEM qualifier, the default number of pointers is established by the ACP_WINDOW system parameter. Otherwise, the value is taken from the /WINDOW qualifier the user specified on the command line.

The LRU limit is the directory preaccess limit. It is a count of the number of directories to be stored in the directory index cache. For a volume managed by an ACP, it is an estimate of the number of concurrent users (that is, the number of directories that will be in use concurrently) on the volume. If the volume is being mounted with the /SYSTEM qualifier, the default number of directory FCBs to be cached is set with the ACP_SYSACC system parameter. Otherwise, the value is taken from the /ACCESS qualifier the user specified on the command line. If the user explicitly specified that no caching was to be enabled for the volume, then the LRU limit is set to 0.

For a volume managed by an XQP, however, the LRU limit is obsolete. Accessed directories are instead managed on a per-buffer-cache basis and are limited by the ACP_DINDX_CACHE system parameter.

The value in the VCB$W_EXTEND field is copied from the HM2$W_EXTEND field in the home block. Although the VCB$W_EXTEND field is set up by MOUNT, it is not used by any other VMS facility (including RMS).

The number of blocks that are allocated to a file when a user extends the file and asks for the system default allocation is taken from the HM2$W_EXTEND field. If the current value is 0, it is set to 5. Otherwise, the value is taken from the /EXTEND qualifier the user specified on the command line.

The index file bitmap size and the maximum number of files are also established by the fields of the home block.

If the user specified the /CACHE=WRITETHROUGH qualifier, the VCB$V_WRITETHRU bit is set to 1. If the /NOCACHE qualifier was specified, the VCB$V_NOCACHE bit is set to 1.

3.3.2.5 Initializing the Prototype Index File FCB

The first step in initializing the FCB of the index file is to read and verify the index file header. If the header is invalid, the alternate index file header is used instead. The address of the header is used to initialize the prototype index file FCB.

The map area of the index file, pointed to by the FH2$B_MPOFFSET field of the home block, is scanned. The file size is calculated from the value in the FCB$L_FILESIZE field, plus the number of retrieval pointers.

In computing the number of retrieval pointers, the type of map pointer is determined, and the size and LBN fields of the pointer are filled in accordingly. The pointer count is incremented, and another pointer is fetched; placement pointers, however, are transparently excluded in the count. For more information on the format of retrieval pointers, see Sections 2.3.3.3.1 through 2.3.3.3.4.

The rest of the fields in the index file FCB are then filled in. The file attributes of the FCB are updated using the attributes of the index file header, but the file size is preserved.

The address of the object rights block (FCB$R_ORB) is noted. The header LBN, file ID, starting VBN, file owner, and file protection are filled in. The **lock basis**, from which a serialization lock is constructed, is extracted from the low-order file number (FCB$W_FID_NUM), the high-order file number (FCB$B_FID_NMX), and the relative volume number (FCB$B_FID_RVN). If the directory bit is not set, add 1 to the volume's directory LRU.

3.3.2.6 Constructing the Prototype Index File Window

After the prototype index file FCB is set up, the prototype index file window is built. The WCB$W_SIZE field is calculated by adding the value specified by WCB$C_LENGTH and the value the user specified with the /WINDOW qualifier on the MOUNT command line. The WCB$V_READ bit is set to 1, allowing read access. Then a window is set up that maps as much as possible of the index file, starting with VBN 3. The presence of the WCB and the FCB causes the index file to be open on the volume when it is mounted.

The map area of the file header is scanned, and retrieval pointers in the window are built until one of the following results occurs:

- The entire header has been scanned.

- The first retrieval pointer in the window maps the desired VBN.

The window is scanned for the starting VBN of the header. If the VBN is contained within the window, the window is truncated so that it maps up to the start of the header exactly.

However, if the starting VBN of the header is not contained in the window, the entire window must be discarded (it is retained in cache, if possible) in preparation for a window turn. However, if the desired VBN precedes the starting VBN of the header, the existing window is the best possible effort.

After the window is initialized, the necessary pointers are set up. The map area is scanned and the retrieval pointers are obtained.

As many new retrieval pointers are built as necessary to describe the window. If the window is full, the entries are shifted up by one until the operation would cause the pointer mapping the desired VBN to shift off the top. Finally, the pointer is built, and is included in the count given by the WCB$W_NMAP field.

3.3.2.7 Reading the SCB

The storage map file header is read, and the starting LBN of the storage bitmap is calculated. The size of the storage bitmap is computed from the volume size and cluster factor because the storage bitmap file is rounded up to the next cluster boundary.

The storage control block is read. The shared file system cannot tolerate failure to read the storage control block because that is where the volume label used for locking is stored.

3.3.2.8 Establishing the Volume Lock

The volume lock is obtained and the volume lock name is established. The resource name used for the volume allocation lock is stored in the VCB.

If the volume is being mounted with the /NOSHARE qualifier, the resource name is a unique node identifier plus a unique device identifier. The node identifier is taken from the global cell SCS$GB_NODENAME and stored in the prototype VCB$T_VOLCKNAM field. The device identifier is taken from the UCB of the device being mounted.

For shared mounts, however, the resource name is the volume label. Because volume labels may change after the volume is mounted, the first process to mount the device for write access writes the volume label used into the SCB$T_VOLOCKNAME field. All other processes mounting the volume read the SCB$T_VOLOCKNAME field for the resource name and write it into the prototype VCB$T_VOLCKNAM field.

The volume allocation lock is then acquired in protected write mode, which is necessary to allow the value block to be written later. If this is a nonshared mount, the system-owned lock (stored in the global cell EXE$GL_SYSID_LOCK) is used as a parent lock to cause the lock to be mastered locally without any cluster message traffic from the distributed lock manager.

A $GETLKI function is also performed on the volume allocation lock to determine the number of locks granted on that resource (that is, the number of VAXcluster nodes that have this volume mounted). This information is later used to determine whether a rebuild operation should be performed on the volume after it is mounted.

If the count of volume locks does not match the count in the storage control block (SCB$W_WRITECNT), the count is updated, and the volume is marked to be rebuilt, if necessary (depending on the flags set in the SCB$L_STATUS2 field). The SCB$L_STATUS2 flags are cleared only upon successful completion of a rebuild, so rebuilds are attempted until the volume is actually rebuilt.

The flags in the SCB$L_STATUS field are set to mark which caches are enabled. If the size of the extent cache has been established, the SCB$V_MAPALLOC bit is set. Likewise, if the size of the file ID cache has been established, the SCB$V_FILALLOC bit is set. If the size of the quota cache has been established and the device is not part of a volume set, the SCB$V_QUODIRTY bit is set. These bits may already be set if the disk has been mounted elsewhere in the VAXcluster with the same caches enabled.

The storage bitmap sequence number in the volume lock value block is incremented to invalidate potential copies in the file system caches. The storage control block is rewritten; if the write fails, the volume is write-locked.

If this is not the first mount for this device, essential mount parameters are checked to make sure they are consistent. The information from the current mount request is compared with the value block of the device lock, which contains information about this device from all nodes in the cluster. This information must be consistent across the cluster, and it includes the following parameters, for example:

- The ownership of the volume
- The protection of the volume
- Whether the volume is locked or enabled for write access
- Whether the volume has been mounted foreign or file-structured

The device and volume lock value blocks are also compared to see if this is the first mount for this device. If the value blocks do not match, another volume of the same name is already mounted in the cluster.

3.3.2.9 Locating the Highest File Number

The index file bitmap is scanned backwards from the end to find the highest file number. The VBN of this file is compared to the index file end-of-file mark. If the EOF is short, the EOF delta is set higher so that the first create operation will update the index file header. If this is not the initial mount of the volume, the index file EOF is copied from the value block.

If this is the first mount of the volume, the storage map is scanned to compute the number of free blocks on the volume, and the VCB$L_FREE field is updated.

3.3.2.10 Allocating the I/O Database Structures

The routine MAKE_DISK_MOUNT performs all of the database manipulation needed to mount a volume. The mode is set to kernel to gain write access to the I/O database.

This routine allocates the real VCB, FCB, and WCB in nonpaged system pool, and links them. The control blocks are all allocated in advance to avoid having to back out of some awkward situations later. The one exception is the AQB, which is either found or allocated by the START_ACP routine.

The first required control block to be allocated is the VCB. The following actions are performed:

- Memory is allocated according to the size indicated by the VCB$C_LENGTH field.
- The constant DYN$C_VCB is written to the VCB$B_TYPE field.
- The UCB is established, and forward and backward links are set up.
- The object rights block (ORB) is established, and forward and backward links are set up.

The index file FCB is the second structure to be constructed. The following actions are performed:

- Memory is allocated according to the size indicated by the FCB$C_LENGTH field.
- The constant DYN$C_FCB is written to the FCB$B_TYPE field.
- The forward and backward links to the window listhead are set up.
- The ACL fields in the ORB are initialized.
- The FCB is inserted as the first element on the volume control block's FCB list.

The index file WCB is the third structure to be constructed. The following actions are performed:

- Memory is allocated according to the size indicated by the WCB$C_LENGTH field and the number of mapping pointers indicated by the WCB$W_NMAP field.
- The constant DYN$C_WCB is written to the WCB$B_TYPE field.
- The WCB is inserted on the file control block's WCB listhead.

The cache block (VCA) for the volume is then allocated. Its size is computed from the cache parameters. The address of the VCA is written to the VCB$L_CACHE field, and the constant DYN$C_VCA is written to the VCB$B_TYPE field.

If the volume is part of a volume set, the volume is attached to the RVT for the set, creating an RVT if one does not exist. The pointers to the VCB and the WCB, as well as a lock basis, are established. The volume set lock is taken out, and the volume set structure name is checked to ensure that it is unique.

Space is allocated for logical name and mounted volume list entries. If a logical name is given in the command, it is assigned to the volume. Otherwise, the logical name is constructed from the volume label.

3.3.2.11 Creating the AQB

At this point, all data blocks except the AQB have been allocated. Before the AQB can be allocated, the volume ownership and protection must be set up in the VCB (in the ORB$L_OWNER field). The default comes from the volume UIC; otherwise, a volume owner is established on the command line by the /OWNER_UIC qualifier. The rest of the data structures are hooked up in the device database.

The VCB pointer in the UCB is set up. The I/O database mutex (IOC$GL_MUTEX) is acquired. Nonpaged dynamic memory for the AQB is allocated and initialized, and the DYN$C_AQB constant is copied into the AQB$B_TYPE field. The mount count is initialized to 1, indicating that a single volume is being handled. The AQB$V_XQIOPROC field is set to 1, indicating that an XQP is being used.

The AQB forward and backward links are set up. The AQB is linked into the system AQB list, headed by the system cell IOC$GL_AQBLIST.

The routine SETUP_BLOCKCACHE is called to allocate dynamic memory and initialize it for use in the buffer cache.

The UCB$V_MOUNTING field is initialized to 1. The address of the AQB is moved into the VCB.

3.3.2.12 Establishing File System Context

The various value block contexts are stored by converting the volume, volume set (if present), and device locks to their system-owned compatible modes.

The device lock may not be present if the device is not cluster-accessible.

In addition, the IO$_MOUNT function is issued to synchronize the file system to the newly mounted volume. The following actions then occur:

- The function decision table (FDT) processing for IO$_MOUNT checks and clears the UCB$V_MOUNTING bit.
- The file system finds the AQB and verifies that the file structure type is one that it supports.
- The file system sets the UCB$V_MOUNTED bit.

MOUNT also creates the **mounted volume list entry** (MTL), or **mount list entry**, and the logical name for the volume. An MTL appears in the job mounted volume list for each volume mounted by the process with either the /SHARE or the /NOSHARE qualifier. In addition, each volume mounted with the /SYSTEM or the /GROUP qualifier has an entry in the systemwide mounted volume list. The list itself is a doubly linked list.

The fields of the mounted volume list entry are shown in Figure 3–11 and are described in Table 3–7.

Figure 3–11: Format of the Mounted Volume List Entry

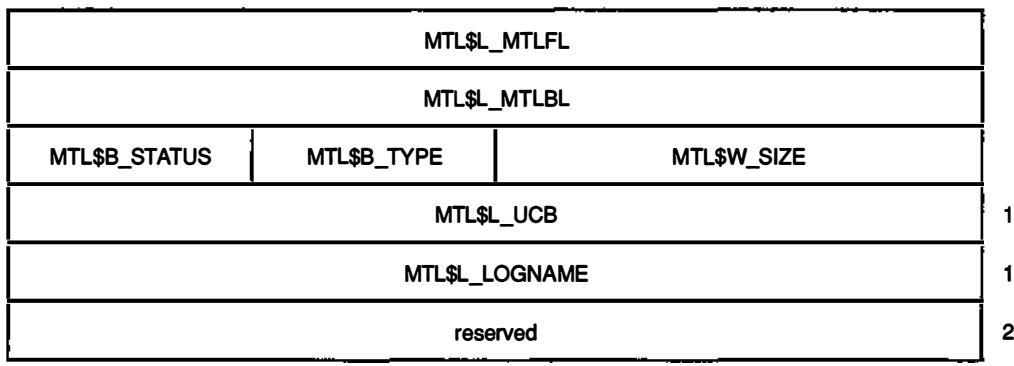


Table 3–7: Contents of the Mounted Volume List Entry

Field Name	Description
MTL$L_MTLFL	Forward pointer to the rest of the entries in the list.
MTL$L_MTLBL	Backward pointer to the rest of the entries in the list.
MTL$W_SIZE	Structure size in bytes.
MTL$B_TYPE	Structure type code. This field contains the DYN$C_MTL type code to identify the data structure as a mounted volume list entry.
MTL$B_STATUS	Status byte. MTL$V_VOLSET is defined within MTL$B_STATUS. This mounted volume list entry is for a volume set. This is bit 24.
MTL$L_UCB	Pointer to device UCB.
MTL$L_LOGNAME	Pointer to the logical name associated with the volume. If this field contains a value of 0, no logical name exists for the volume.

3.3.3 Processing a Volume Set

A **volume set** is a collection of related volumes that is normally treated as a single logical device. Information about a volume set is maintained in the relative volume table (RVT), which is a dynamic structure in the I/O database that is allocated from nonpaged pool. The format of the relative volume table is discussed in Section 3.3.1.5.

Each volume in a volume set contains its own Files–11 structure; however, files on the various volumes in a volume set may be referenced with a relative volume number that uniquely determines on which volume in the set the file is located.

A volume set has a structure name associated with it, which is a string of up to twelve ASCII characters which identifies the volume set. The characters in the structure name should not include control characters or the delete character, and the structure name cannot be null.

The volume label of each of the volumes making up the set must be unique within the set, and must be different from the structure name. The first relative volume of the set contains a volume set list file (VOLSET.SYS, located in the master file directory) which lists the volume labels of all the volumes in the set, thus associating volume labels with relative volume numbers. Each volume is identified as being part of the set by carrying the structure name, its volume label, and its relative volume number.

Volume set processing differs from single-volume processing in two main areas:

- Space management—When a file is created, the file is located on the volume with the most space, unless the user explicitly specifies the placement. Once created, a file is always extended on the volume on which it resides unless the volume is full or the user specifies the placement.
- Root volume—The root volume must be mounted for any of the remaining volumes in the set to be referenced.

3.3.3.1 Creating a Volume Set

Two or more disk volumes may be bound into a volume set using the MOUNT /BIND=volume-set-structure-name command. The volumes specified in the volume-label list are assigned relative volume numbers based on their position in the label list. The first volume specified becomes the root volume (relative volume 1) of the volume set.

Also, a volume may be added to an existing volume set that is already mounted by using the MOUNT/BIND command. The /BIND qualifier needs to be specified the first time the volume set is created or when a new volume is added. On subsequent mounts, the MOUNT command uses the information (for example, the structure name and the relative volume number) recorded in the home block to form the volume set.

3.3.3.2 Mounting a Volume Set

The following steps are performed for each volume set member when a volume set is created or mounted:

- The volume set structure name is established.

  If the relative volume number field in the home block (the HM2$W_RVN field) contains a value of 0, indicating that this volume has never been a part of any volume set, then the structure name is established from the /BIND qualifier on the MOUNT command line.

  If the HM2$W_RVN field contains a nonzero value, then the structure name in the HM2$T_STRUCNAME field is used.

  When the /BIND qualifier is specified and the HM2$W_RVN field contains a nonzero value, then the structure names specified in the /BIND qualifier and the HM2$T_STRUCNAME field in the home block must match.
- The volume is mounted as a single volume.
- A routine is called to enter this volume into the relative volume table. This routine finds the RVT of this volume set by its structure name.

If such an RVT does not exist (that is, this volume is the first volume of the set to be mounted), an RVT is created, and the relative volume number for this volume is set to 1 if its RVN was 0.

If an RVT already exists (that is, this volume is not the first to be mounted), this volume is entered in the RVT. When a new volume is added to the set, the relative volume number is set to the previous number of volumes in the set, plus 1. For an existing volume already bound to the volume set, the value in the HM2$W_RVN field is used as the relative volume number.

- When a volume set is created, or when a volume is added to an existing volume set (with the MOUNT/BIND command), the new volume must be bound into the volume set. Its volume label is entered into the volume set list file, which is the VOLSET.SYS file in the MFD on the root volume. For this reason, the root volume must be mounted first (or be already mounted) in a /BIND operation. Also, the home block of this new volume must be updated to reflect that it is a member of the volume set. That is, the HM2$T_STRUCNAME field and the HM2$W_RVN field are updated to reflect this volume's membership in the volume set.

3.3.4 Rebuilding the Bitmap and Disk Quota Files

If a disk volume has been improperly dismounted (for example, as a result of a system failure), the index file bitmap, the allocation bitmap, and the quota file must be rebuilt in order to recover caching contents that were enabled on the volume at the time of the dismount. By default, MOUNT attempts the rebuild. The rebuild may consume a considerable amount of time, mainly depending upon the number of files on the volume.

The following three types of caches may be enabled on a volume:

- Extent cache—preallocated free space. Blocks are allocated from the volume and marked in the storage bitmap as being in use.
- File ID cache—preallocated file numbers. File IDs are preallocated from the index file and marked in the index file bitmap as being in use.
- Quota cache—disk quota usage.

Sections 4.2.8 through 4.2.10 contain more information on these caches.

If both extent caching and file ID caching were enabled, the rebuild time is directly proportional to the greatest number of files that ever existed on the volume at one time. If disk quota caching was enabled, additional time to handle quota processing may be required.

If none of these caches was in effect, then the rebuild is not necessary and does not occur.

When a volume is rebuilt, the ACP control function IO$_ACPCONTROL is issued to lock the volume against modifications while the rebuild is taking place. The FIB$C_LOCK_VOL constant is specified in the FIB$W_CNTRLFUNC field. The ACP control function sets the VCB lock bit in the VCB$W_ACTIVITY field.

Virtual memory for I/O buffers is allocated with a call to LIB$GET_VM.

The index file INDEXF.SYS is opened with the ACP function IO$_ACCESS (or the ACP subfunction IO$M_ACCESS). The home block is read and validated, and the volume characteristics are established, including the VBN offset for the file headers of the volume, the cluster factor of the volume, the VBN offset of the index file bitmap, and the end of the index file. The INDEXF.SYS file is then deaccessed by the ACP function IO$_DEACCESS.

The storage bitmap file is accessed so that the SCB can be checked to see if a rebuild is still necessary. The flags of the SCB$L_STATUS2 field are set when the volume is mounted if the following two conditions exist:

- The corresponding flags are set in the SCB$L_STATUS field.
- The current count of VAXcluster nodes enabled for write access to the volume (contained in the SCB$W_WRITECNT field) does not match the current number of outstanding locks. This condition indicates that caching was enabled on the volume and that the volume was improperly dismounted.

If the volume is part of a volume set, the index file is accessed for each volume in the set to obtain the characteristics of each volume.

Two types of rebuild operations may occur: conditional or unconditional. If the rebuild is conditional (that is, if the QUODIRTY or MAPDIRTY bits are set in the SCB, meaning that the quota file or storage map are only partially updated), the quota file is checked to see if it needs to be rebuilt; it is not rebuilt unless it was active. If the quota file has to be rebuilt, the storage bitmap file and the allocation bitmap are also rebuilt. If nothing needs to be rebuilt, the volume lock is released and the rebuild is finished.

If the rebuild is not conditional (that is, the volume will be rebuilt regardless), the FIB$C_ENA_QUOTA constant is moved into the FIB$L_CNTRLFUNC field to enable the quota file. The quota file is then scanned from beginning to end to build the usage table. The usage table is a chained hash table consisting of entries for all the quota file records that show zero blocks in use. The key to the hash table is the UIC that is to be stored there. The hash function is a simple modulus function on the UIC of the file in question with the number of entries in the table. The overflow caused by collisions is handled by chaining each bucket. The quota file is then deaccessed.

The main reasons for having conditional and unconditional rebuilds is because the System Management Utility (SYSMAN)[1] and the SET VOLUME/REBUILD command specify rebuild operations.

The bitmap file is opened for each volume in the volume set, and all the file headers are read. The SCB is also read to obtain the volume size and cluster factor. From these, the size of the allocation bitmap is calculated. The allocation bitmap is then initialized to show all the space available on the volume. A set bit in the bitmap indicates a free block on the volume.

The index file is opened. Virtual memory for a working copy of the index file bitmap is allocated. The old index file bitmap is read into a buffer.

All the virtual blocks in the index file are scanned. The file headers, starting with the MFD, are read, and each is verified to ensure that it is valid. The verification includes the following checks:

- The structure level contained in the FH2$B_STRUCLEV must equal 2.
- The area offsets and the retrieval pointer use counts must be consistent.
- The file number (comprised of the FH2$W_FID_NUM and FH2$B_FID_NMX fields) cannot equal 0.
- The header checksum is calculated and validated.
- The header file number (FH2$W_FID_NUM) and file sequence number (FH2$W_FID_SEQ) are compared to the file number (FID$W_NUM) and file sequence number (FID$W_SEQ).

If the file header is valid, the number of blocks it occupies is computed and then charged to the owner UIC. The blocks in use are marked in the storage bitmap, and, if quotas are being rebuilt, are entered in the quota hash table under the owner UIC of the file.

However, if the file header is not valid, a check is made to see if it is marked as being in use in the index file bitmap. If it is, the sequence number of the file ID is incremented and written. The header is then marked as being free in the index file bitmap.

If an I/O error occurs while a header is being read, an invalid header with a random sequence number is written to the header slot. This action is to prevent a valid file header from reappearing later, if the I/O error happens to be transient.

After all the headers have been accounted for, all the unreferenced bits past the end of the index file are cleared. The index file bitmap is written back to disk. The virtual memory for the working copy of the index file bitmap is released.

[1] SYSMAN includes the functions of the Disk Quota Utility (DISKQUOTA), which operated as a standalone utility in VMS Version 4.6.

The new storage bitmap is written back to disk and any working memory is also released. The pertinent flag bits (SCB$V_MAPDIRTY2, SCB$V_MAPALLOC2, and SCB$V_FILALLOC2) are cleared in the SCB.

The number of free blocks (that is, the volume size) is updated in the VCB.

At this point, the entire volume set has been scanned, and a table of total disk usage (the usage table) for the volume exists. Each table entry is used to update the corresponding quota file entry in QUOTA.SYS. The update consists of two passes:

1. Existing entries are updated and the volume is unlocked.
2. New quota file records are created for UICs that have blocks in use but have no corresponding quota file entries. Because the quota file may have to be extended, this action must be done after the volume is unlocked.

The SCB$V_QUODIRTY2 flag bit must be updated in the SCB, so the storage bitmap file is opened for write access. The flag bit is cleared, and the SCB is then written back to disk. The storage bitmap file is then closed.

At this point, the volume has been rebuilt. Any remaining virtual memory is released, and the rebuild exit handler is canceled.

3.4 Dismounting a Volume

The volume dismount operation is the complement of the volume mount. Like the Mount Utility, the Dismount Utility (DISMOUNT) is a privileged shareable image. Its chief function is to mark a volume for dismount.

A volume will not be dismounted[1] if the transaction count is greater than 1, which can be caused by any of the following reasons:

- Paging files, swapping files, or images have been installed on the volume.
- Devices are spooled to the volume.
- Secondary page or swap files are resident on the volume.
- Files are open on the volume.

When a dismount operation for a Files–11 volume is requested, the volume is only marked for dismount. Once the volume is marked, the final cleanup operations (for example, the deallocation of the FCBs and the VCB) are done by the file system when the last file on the volume is closed. Therefore, there is a delay from the time the volume is marked for dismount to the time the volume is actually dismounted.

[1] This can be overridden by the DISMOUNT/OVERRIDE=CHECKS command.

The asynchronous nature of the volume dismount requires that three different components of the system be involved in the dismount procedure:

Component	Function
Dismount Utility	Prepares the volume to be dismounted
IOC$DISMOUNT routine	Handles device-independent dismount processing
File system	Handles the actual dismount of the volume

3.4.1 Beginning the Dismount Procedure

A volume dismount may be triggered by two events:

- If a process explicitly issues a dismount request with the DCL DISMOUNT command or the $DISMOU system service.
- If a top-level process is deleted, then all volumes that were mounted either privately or shareable by this job are implicitly dismounted.

When a dismount operation is requested, the dismount image is activated. The main module of the Dismount Utility is DISMOU, located in the DISMOU facility. This module includes the SYS$DISMOU routine, which contains the basic logic of the DISMOUNT command.

3.4.1.1 Preparing the Volume to be Dismounted

In general, when a dismount is requested, certain information about the device has to be acquired. For example, the mounted volume list entry for this particular device is retrieved, and the name of the device to be dismounted is determined. If a logical name is associated with the physical device name descriptor, it is first translated, and then a channel is assigned to the device using the $ASSIGN system service. A channel is needed for two reasons:

- The device UCB address is needed, and it is contained in the CCB.
- The assigned channel acts as an interlock to prevent premature deallocation of the VCB.

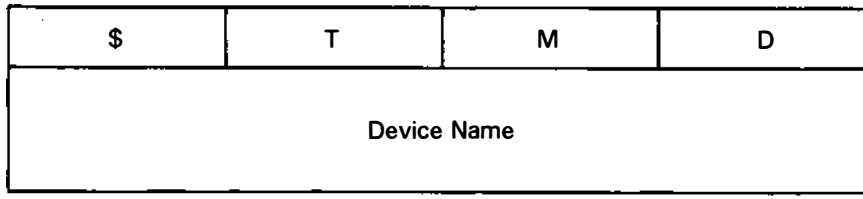
The user is granted the BUGCHECK and EXQUOTA privileges using the $SETPRV system service.

To prevent race conditions between simultaneous dismounts on the same volume, the **dismount interlock** on the device is obtained and taken out in exclusive mode. This lock has the following form:

```
DMT$<allocation-class-device-name>
```

The allocation class device name is established using the $GETDVI system service. Figure 3–12 shows the format of the dismount interlock.

Figure 3–12: Format of the Dismount Lock



ZK-9730-HC

3.4.1.2 Validating the Volume Characteristics

The routine MAKE_DISMOUNT is called to dismount the volume. This routine does the kernel-mode validation and initial setup of the dismount operation. The following device characteristics, returned by the $GETDVI system service, are checked to ensure that the volume can be dismounted:

Device Characteristic	Description
DEV$V_FOD	The device is file-oriented. If this bit is not set, a status of SS$_NOTFILEDEV is returned.
DEV$V_MNT	The device is properly mounted. See the description of the DEV$V_DMT bit for more information.
DEV$V_DMT	If this bit is set, the device is in the process of being dismounted.
	If this bit and the DEV$V_MNT bit are clear, the device is idle (that is, dismounted). In this case, a status of SS$_DEVNOTMOUNT is returned.
	If this bit is clear and the DEV$V_MNT bit is set, the volume is mounted.
DEV$V_AVL	The device is available for use. If this bit is not set, a status of SS$_DEVOFFLINE is returned.

The primary and secondary device characteristics must match; otherwise, the device may be spooled.

The UCB and the VCB addresses for the channel are obtained. If this is a volume set, the RVT address is also obtained. The address of the channel comes from the global cell CTL$GL_CCBBASE, which contains the base address of the CCB table. The address of the UCB comes from the CCB$L_UCB field, and the address of the VCB comes from the UCB$L_VCB field.

The job mounted volume list is searched for entries of the volume; if found, they are removed and the dismount procedure may proceed. If none is found, the system mounted volume list is searched for any volumes that were mounted with the /GROUP or /SYSTEM qualifiers. Dismounting these volumes requires the appropriate privilege. If the volume is mounted by the current process and the dismount request is a normal[1] one, no privilege checks need to be made.

3.4.1.3 Checking Privileges

In the course of verifying whether the dismounter has the proper privileges to dismount the volume, various conditions are checked for. The four main conditions are as follows:

- If the volume was mounted privately
- If DISMOUNT/ABORT or /CLUSTER was specified
- If the volume was mounted privately by another process
- If the volume was mounted for group or system access

3.4.1.3.1 Checking for a Private Mount

The process mount list is searched for a privately mounted volume if the /ABORT or the /CLUSTER qualifier was not specified on the DISMOUNT command line. This is done by obtaining the address of the job information block from the PCB$L_JIB field in the current process control block whose address is stored in the scheduler database. The address of the process or jobwide mount list can then be obtained from the JIB$L_MTLFL field.

The I/O mutex IOC$GL_MUTEX is then acquired to synchronize operations while the entries of the mounted volume list are searched to see if the volume is privately mounted.

The mounted volume list is searched for the entry representing the desired UCB (the MTL$L_UCB field). If the MTL$B_TYPE field does not contain the constant DYN$C_MTL, an error is returned.

After the search concludes, the I/O database mutex is released. If a mounted volume list entry was found, a success status is returned; no privilege is required to dismount a private volume. At this point, a privately mounted volume has been properly dismounted.

---

[1] /ABORT, /OVERRIDE, /GROUP, or /SYSTEM was not specified on the command line.

3.4.1.3.2 Checking for a Volume Mounted with /ABORT or /CLUSTER

If no MTL was found in the process mount list or if the process mount list was not searched because the /ABORT or the /CLUSTER qualifier was specified on the DISMOUNT command line, then the system mount list is searched. The address of the systemwide mounted volume list is contained in the global cell IOC$GQ_MOUNTLST.

To determine if the process has the necessary privilege to dismount the volume, the privilege mask contained in the PHD$Q_PRIVMSK field pointed to by the global cell CTL$GL_PHD is obtained. The address of the VCB contained in the UCB$L_VCB field is also obtained to determine whether the volume was mounted for group or systemwide access.

3.4.1.3.3 Checking for a Volume Mounted Privately by Another Process

If an MTL is not found in the system mount list, then the volume must be mounted privately by some other process. In order for the current process to dismount the volume, the process must meet the following two conditions:

- Must have specified the /ABORT qualifier on the DISMOUNT command line
- Must either own the volume or have the necessary privilege to override volume protection (VOLPRO)

The DISMOUNT/ABORT command allows a user to dismount a volume that is owned by a different process. It is useful, for example, when a volume is mounted shareable (that is, many processes have MTLs for the volume). /ABORT also performs the following tasks:

- Cancels pending I/O
- Cancels mount verification

The address of the object rights block (ORB) is obtained from the UCB in order to determine which process owns the volume. The I/O mutex is then released. The UIC of the current process is obtained from the PCB$L_UIC field in the PCB of the current process. A status value of SS$_NOPRIV is returned if any of the following conditions are met:

- The DMT$V_ABORT bit is set.
- The UIC does not match the UIC contained in the ORB$L_OWNER field.
- The dismounting process does not have VOLPRO privilege.

3.4.1.3.4 Checking for a Volume Mounted for Group or System Access

If an MTL was found in the system mount list, the following two conditions must exist:

- The volume must be mounted for group or system access. The VCB$V_GROUP bit is set if the volume has been mounted with the /GROUP qualifier.
- The current process must have the correct privileges necessary to dismount the volume.

If the PRV$V_SYSNAM bit is set in the privilege mask for the process, the process has SYSNAM privilege, which means that the process can dismount volumes owned by any group, and no further checking needs to be done.

If the process has the PRV$V_GRPNAM bit set in its privilege mask (meaning that it has GRPNAM privilege), the ownership of the volume must be checked to see if the current process is in the same group as the volume owner. This is done by determining which group owns the logical name table that contains the logical name for the volume.

If there is no logical name associated with the volume (the MTL$L_LOGNAME field contains a value of 0), it is assumed that the process is in the correct group. However, if there is a logical name associated with the volume, then the logical name mutex is obtained, and the MTL$L_LOGNAME field is used to obtain the address of the logical name block. The address of the logical name table is obtained from the LNMB$L_TABLE field in the logical name block, and the address of the object rights block is obtained from the LNMTH$L_ORB field in the logical name table.

The UIC or the volume owner is then obtained from the ORB and the logical name mutex released. The upper 16 bits of the volume owner UIC (the UIC group number) are then compared with the UIC group number of the current process (contained in the PCB$W_GRP field in the PCB of the current process). If they do not match, the I/O mutex is released, and a status value of SS$_NOGRPNAM is returned.

If the VCB$V_SYSTEM bit is set, the volume has been mounted for system access. If the PRV$V_SYSNAM bit is not set in the privilege mask, then the I/O database mutex is released and a status value of SS$_NOSYSNAM is returned.

After the process has been checked to see if it has the necessary privileges to dismount a volume that was mounted for group or system access, the I/O mutex is released. One last check compares the UIC of the device being dismounted with the UIC of the system disk.

A system disk cannot be dismounted; otherwise, all I/O to the system disk, such as image activation, fails. The global cell EXE$GL_SYSUCB contains the address of the UCB of the system device. If the device being dismounted is a system disk, the dismount operation fails with a value of DISM$_SYSDEV.

3.4.1.4 Setting Up the Local Mounted Volume Database

The routine SETUP_MTL sets up a local mounted volume database by collecting the appropriate mount list entries from the system's mounted database. How it is set up depends on whether a normal dismount or DISMOUNT/ABORT was specified.

If DISMOUNT/ABORT was specified (the DMT$V_ABORT bit is set) the I/O database must be scanned. This requires raising IPL to IPL$_SYNCH (IPL 8) so that systemwide data structures can be searched. Also, no page faults can be incurred while at IPL$_SYNCH, so the pages are explicitly locked in memory.

The PCB of the null process is obtained from the scheduler database (to be able to distinguish a nonnull process), and the I/O database mutex is acquired. The scheduler database is searched for valid processes that have MTLs for the mounted volume.

The IPL is lowered to IPL$_ASTDEL (IPL 2—the highest level at which page faults are permitted) because mounted volume lists are located in paged pool. This can be done because the existence of a mounted volume list entry means that this process will not be deleted until the I/O database mutex is released. At this point, the jobwide mount listhead JIB$L_MTLFL is used to find the correct entry.

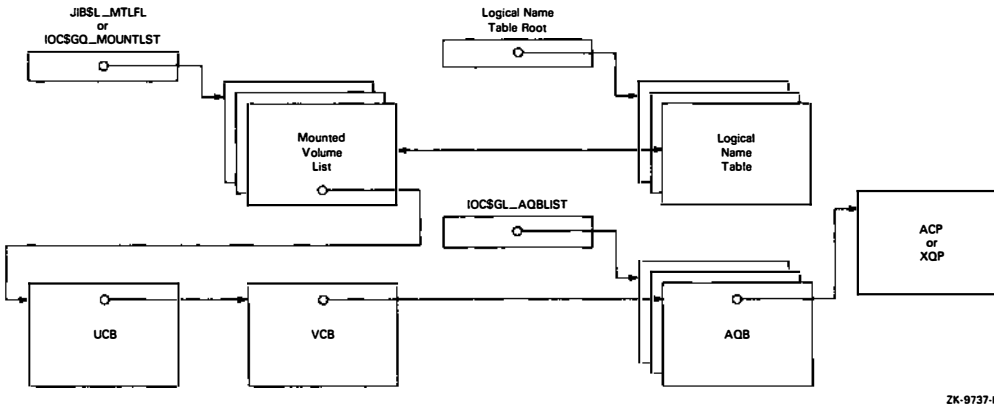
Setting up the local mounted volume database is much more simple in the case of a normal dismount. If a normal dismount was specified (the DMT$V_ABORT flag is not set), the scheduler database is used to find the address of the JIB contained in the PCB$L_JIB field. After the I/O database mutex is acquired, the jobwide mount listhead JIB$L_MTLFL is used to find the correct entry.

All the mounted volume list entries for the current process must be found so that the local mounted volume database can be set up. Figure 3–13 shows the relationship of the data structures in the mounted volume database.

The MOVE_MTL routine removes all the appropriate MTLs from the mounted database and moves them to the local mounted volume database. If the volume is a member of a volume set, then the routine loops for each volume. If the DMT$V_UNIT flag is set, however, the requested UCB points to a single volume rather than to a member of a volume set. The address of the RVT is obtained from the VCB$L_RVT field, and that of the UCB is obtained from the RVT$L_UCBLST field.

If an entry is found, it is removed from the old list and inserted into the new list representing the local mounted volume list database. After all the entries have been located, the I/O database mutex is released, and the system dismount routine is called.

Figure 3–13: Mounted Volume Database



3.4.2 Device-Independent Dismount Processing

In the second step of the volume dismount procedure, the device-independent dismount routine in the VAX/VMS executive—the IOC$DISMOUNT routine—is called to dismount the volume. This routine is called when any device is dismounted, regardless of whether the volume is mounted Files–11 or as a foreign volume. It performs some of the device-independent dismount operations for the indicated MTL entry.

To begin with, the routine performs the following three actions:

- The logical name associated with the volume is deallocated.
- The mount list entry is deallocated.
- The mount count in the VCB (the VCB$W_MCNT field) is decremented.

  If the mount count is nonzero (for example, more than one process mounted the volume with the /SHARE qualifier), this routine is complete, and control is returned to the caller.

When the mount count reaches zero, the device is marked for dismount. At this point, the volume's mount verification bit (VCB$V_MOUNTVER) is cleared to disable future mount verification on the volume.

A channel is assigned to the device. For nonforeign devices, an ACP control function with the dismount subfunction is issued to the file system (whether the file system is the XQP, the F11AACP, or the MTAACP). When the ACP control function completes, the channel is deassigned, and control is returned to the caller.

3.4.3 Final Dismount Processing

The initial file system dismount occurs when the file system receives an ACP control function with the dismount subfunction.

This routine in the XQP performs the following actions:

- The FID cache, the extent cache, and the quota cache are flushed.
- The SCB$W_WRITECNT field in the storage control block is decremented, indicating that the volume has been properly dismounted.

Any subsequent file operations are done without caching to preserve the integrity of the volume.

Final file system dismount processing occurs after two conditions have been met:

- The last file is closed on the volume.
- The last queued file system function has been processed.

In the XQP's I/O completion routine, the CHECK_DISMOUNT routine is called to check if the volume is marked for dismount. If it is, and the volume transaction count in the VCB$W_TRANS field is 1, then the volume is dismounted immediately. The UCB$V_DISMOUNT bit in the device UCB is set to stop further activity.

CHECK_DISMOUNT performs the following tasks:

- Makes an error log entry to record this dismount operation and send it to the error logger.
- Issues an available (or an unload) I/O function to the driver to clear the drive's volume valid status (or to unload the device).
- Raises the device lock to protected write mode, if the volume was mounted shareable, so the lock can be written back later. If the mount was private, the device lock is already at exclusive mode.
- Marks the volume as dismounted by clearing the following bits:

  DEV$V_MNT Device is mounted.

  DEV$V_DMT Device is marked for dismount.

  DEV$V_SWL Device is software-writelocked.

- Decrements the device reference count in the UCB$W_REFC field.
- Clears the device protection fields in the ORB.
- Disconnects the VCB from the UCB by clearing the pointer to the VCB (the UCB$L_VCB field).
- Decrements the mount count on the AQB (the AQB$B_MNTCNT field). If this mount count has a value of 0, this AQB will be deallocated later.

- Deallocates all FCBs.
- Deallocates the FID cache, the extent cache, and the quota cache, and dequeues the corresponding cache locks.
- Dequeues the volume allocation lock.
- Deallocates the VCB.
- Demotes the device lock to the appropriate mode. If this is the final dismount in the cluster, the device lock value block is zeroed.
- Deallocates the device if the device is marked as deallocate-on-dismount.
- Deallocates the AQB, if necessary. The buffer cache associated with the AQB is also deallocated.

At this point the volume is completely dismounted.

Chapter 4

Cache Processing on a Single Node

Buffer n. *[Origin obscure: possibly Italian* buffo *"farcical, comic" or Latin* bufo *"a toad."]* **1** *A region between two devices designed to distort or, if possible, prevent the flow of data in either direction.* **2** *An old, greasy, and abrasive rag used to clean tape heads.*
Stan Kelly-Bootle

Cache *A very expensive part of the memory system of a computer that no one is supposed to know is there.*
Jeff Pesis

Outline

4.1 Introduction

Cache processing is a part of volume structure processing because it is an efficient way of transferring data from disk to memory and back again. The contents of the cache buffers are copies of the corresponding disk blocks (with the exception of the directory index cache).

The file system manages its cache buffers as an LRU, or **least recently used**, cache whose purpose is to retain in memory the buffers corresponding to the disk blocks that the file system has most recently referenced. In this way, the data does not have to be transferred from disk after it has already been read from disk.

A major task of the XQP I/O buffer cache is to provide a shared, systemwide cache in a multithreaded, procedure-based environment. Each node maintains a systemwide I/O buffer cache. All XQP I/O is performed to the buffers in the cache.

To maintain coordinated access to the file structure and to the buffer cache, the XQP uses the **distributed lock manager**. Because the file structure components (such as file headers, bitmaps, and directories) are themselves contained in or associated with files, the components are generally synchronized with locks corresponding to their associated files. For example, both the header and all the data blocks of a directory are synchronized under a single lock based on the directory's file ID. Each data block in the buffer cache is therefore identified by the lock under which it is read.

4.2 Buffer Initialization and Allocation

The XQP uses a systemwide (single-node) I/O buffer cache allocated from paged pool. The Mount Utility qualifiers are used to control buffer cache creation when a disk is mounted. By default, all mounted volumes share the same buffer cache that is allocated when the system disk is mounted during the boot process. If the system parameter ACP_MULTIPLE is set, a new cache for each different device type will be created.

A separate, private I/O buffer cache can be specified with the MOUNT qualifier /PROCESSOR=UNIQUE. A specific I/O buffer cache can be specified with the /PROCESSOR=SAME:mntdev qualifier, where **mntdev** is the name of an already mounted device. In VMS Version 3, these qualifiers created unique ACP processes for concurrency and caching. In Version 4, they create unique caches, and concurrency is provided by the XQP design.

For most systems, increasing the size of the system default cache is better than creating multiple caches (that is, one 800-block cache is more adaptive than two 400-block caches). In rare circumstances, unique caches might be useful to prevent activity on one set of volumes from flushing caches on a second set of volumes.

The SHOW DEVICE/FULL command shows the maximum buffers in the file system cache and the size of the cache in blocks. This number should approximate the sum of the various ACP_xxxCACHE system parameters.

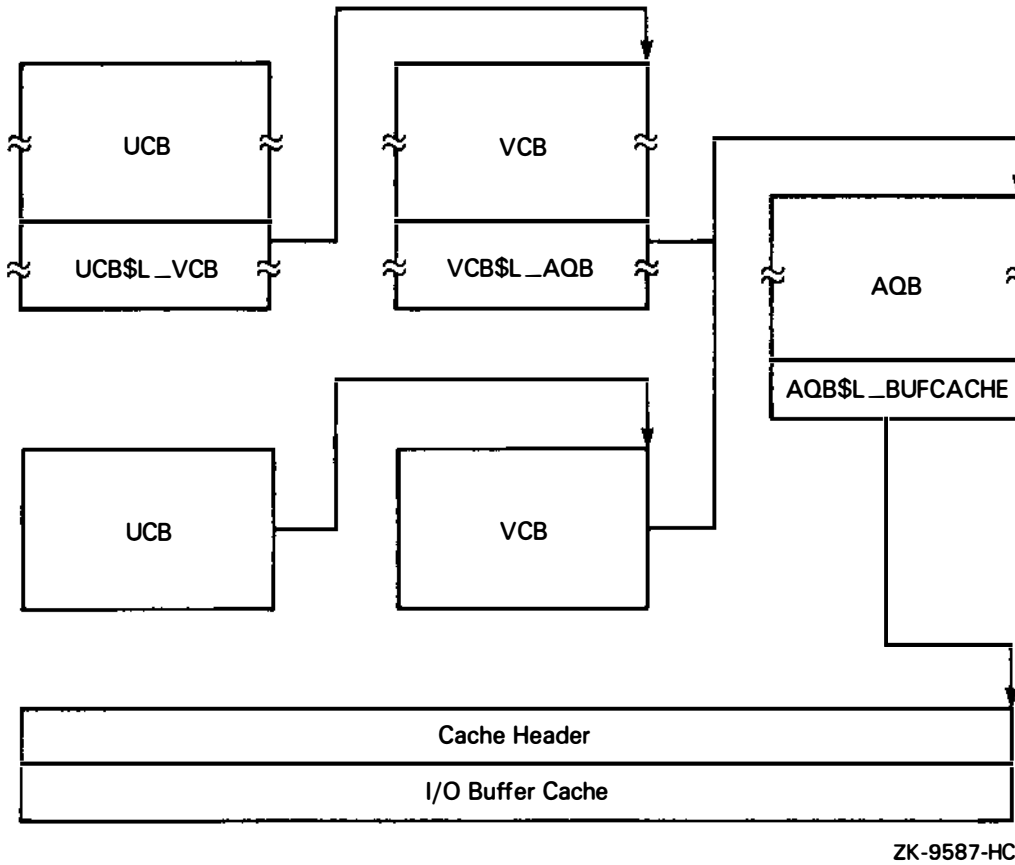
If an attempt is made to allocate a separate cache but the allocation fails because of a lack of sufficient contiguous space in paged pool, MOUNT will try to allocate a minimal size cache instead. If the minimal size cache can be allocated, a reduced cache message (REDCACHE) will be issued, and the volume will be mounted successfully. However, performance will be greatly degraded because, due to the lack of available buffers, only one request at a time can be processed. If the minimal cache allocation attempt fails, an error is returned to the user.

4.2.1 Layout of the I/O Buffer Cache

The cache for a given mounted device is found by following the UCB$L_VCB pointer to the VCB, then the VCB$L_AQB pointer to the ACP queue block, and finally the AQB$L_BUFCACHE pointer to the cache header. There is a single AQB for each buffer cache. However, multiple VCBs may (and usually do) point to a single AQB. The file system always uses the AQB to find the correct cache.

Figure 4–1 shows the how the structures of the I/O database point to the I/O buffer cache.

Figure 4–1: Finding the XQP I/O Buffer Cache



The buffer cache itself consists of various areas:

- **Fixed overhead area**—A fixed area called the **cache header** containing the addresses and the sizes of the following variable areas. The cache header, or **F11BC structure**, also contains the queue headers discussed in Section 4.2.2.

- **Buffer descriptor array**—A variable area containing an array of buffer descriptors, or **BFRD** structures. These structures describe what disk block a given buffer belongs to (by LBN and UCB address), whether it is valid or modified or being used, and what type of buffer it is. It also has an index to its associated lock descriptor.

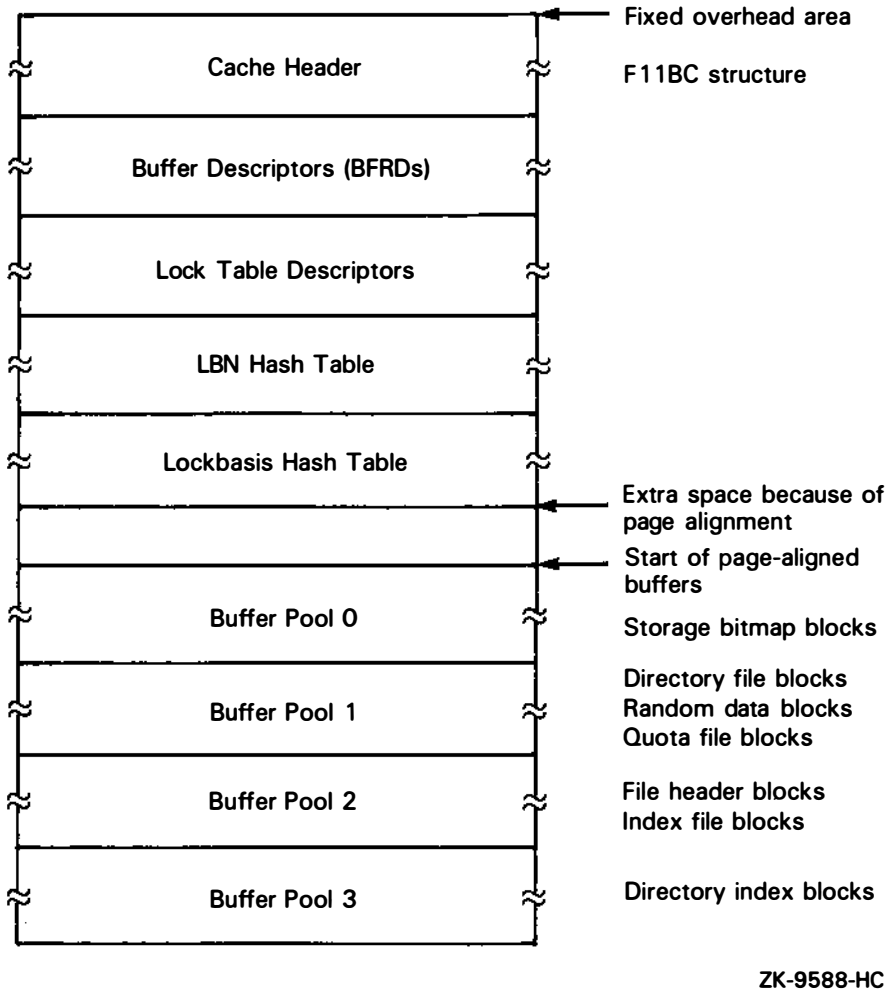
- **Lock descriptor array**—A variable area containing an array of lock descriptors, or **BFRL** structures. These structures describe the locks associated with the buffers in the cache.

- **Buffer LBN hash table**—A variable area containing an array of word indexes into the BFRD array. It tends to reduce the amount of time required to search the cache to determine if a given LBN is already in the cache (over what a linear search of all the descriptors would involve).
- **Lock basis hash table**—A variable area serving a similar function to the LBN hash table. It allows a relatively quick search of the BFRLs to determine if one already exists for a given lock basis. A **lock basis**, consisting of a file number and an RVN, is a unique way of locating a file on a volume set. It is essentially another representation of the file ID, where the sequence number is irrelevant (and is thus omitted). The lock basis becomes a component of the lock resource name for its associated lock.
- **Array of page-aligned buffers**—Buffer pools of variable size. The number of pages allocated for each of the pools is taken from the active values of the ACP system parameters. Caches exist for the following types of blocks:

Block Type	Minimum Size	Location
Storage bitmap blocks	1 block	Pool 0
Directory index blocks	1 block	Pool 3
Directory data blocks	2 blocks	Pool 1
Disk quota file blocks	2 blocks	Pool 1
Random data blocks	2 blocks	Pool 1
File header blocks	3 blocks	Pool 2
Index file bitmap blocks	3 blocks	Pool 2

Figure 4–2 shows the layout of the XQP I/O buffer cache.

Figure 4–2: Contents of the XQP Buffer Cache

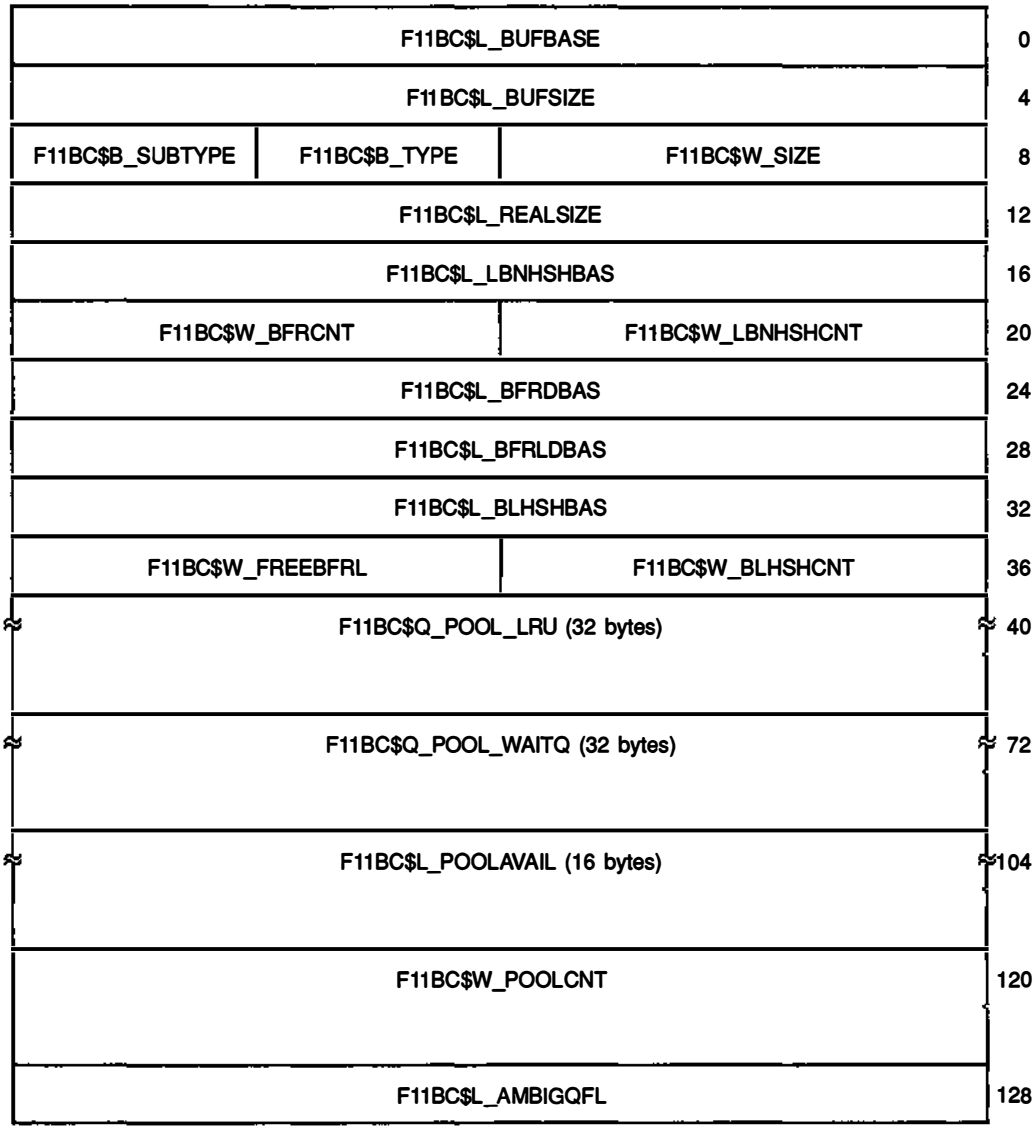


4.2.2 XQP Cache Header

The cache header, represented by the prefix **F11BC$**, contains pointers to the variable areas that follow it. The cache header and the buffers are each allocated as a single contiguous portion of paged pool. However, the descriptor area may be allocated separately from the buffers. The total area occupied by the cache header is about 10% of the area occupied by the buffers.

The fields of a cache header are shown in Figure 4–3 and are described in Table 4–1.

Figure 4–3: Format of the Cache Header



(continued on next page)

Figure 4–3 (Cont.): Format of the Cache Header

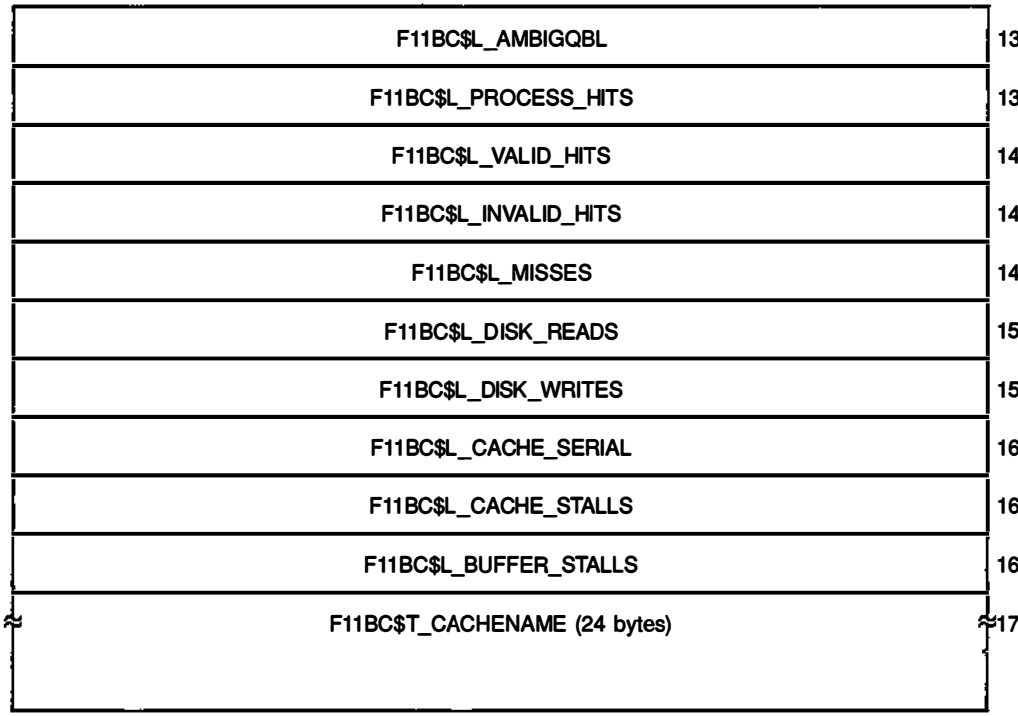


Table 4–1: Contents of the Cache Header

Field Name	Description
F11BC$L_BUFBASE	Base address of the buffer area.
F11BC$L_BUFSIZE	Size of the buffer area in bytes.
F11BC$W_SIZE	Standard size field. This field contains the size of the block. However, because the total size of the buffer cache may exceed 65Kb, the contents of this cell may not reflect the true size of the structure.
F11BC$B_TYPE	Standard (VMS control block) type field. This field contains the DYN$C_F11BC constant.
F11BC$B_SUBTYPE	Standard subtype field.

(continued on next page)

Table 4–1 (Cont.): Contents of the Cache Header

Field Name	Description
F11BC$L_REALSIZE	Size of memory allocated for the whole cache structure.
F11BC$L_LBNHSHBAS	Pointer to the beginning of the LBN hash table.
F11BC$W_LBNHSHCNT	Count of entries in the LBN hash table.
F11BC$W_BFRCNT	Total number of buffers.
F11BC$L_BFRDBAS	Base address of the buffer descriptor area.
F11BC$L_BFRLDBAS	Base address of the buffer lock descriptor area.
F11BC$L_BLHSHBAS	Base address of the buffer lock hash table.
F11BC$W_BLHSHCNT	Number of entries in the buffer lock hash table.
F11BC$W_FREEBFRL	First free buffer lock descriptor in the chain.
F11BC$Q_POOL_LRU	Array of quadword LRU queue headers for each buffer pool. Buffers are arranged in the queues in least recently used order.
F11BC$Q_POOL_WAITQ	Array of quadword cache wait queue headers for each buffer pool. IRPs are inserted in a queue if the process runs out of credits from a particular pool.
F11BC$L_POOLAVAIL	Number of available buffers in each of the buffer pools.
F11BC$W_POOLCNT	Count of buffers in each of the buffer pools. This field is composed of four word subfields (that is, 8 bytes), each of which represents the buffer count in the pool with which it is associated, as Figure 4–4 shows.
F11BC$L_AMBIGQFL	Ambiguity queue forward link.
F11BC$L_AMBIGQBL	Ambiguity queue backward link.
F11BC$L_PROCESS_HITS	In-process buffer hits. This field counts the number of times a buffer was reused from the in-process list.
F11BC$L_VALID_HITS	Valid buffer cache hits. This field counts the number of times a valid buffer is successfully found in the cache.
F11BC$L_INVALID_HITS	Invalid buffer cache hits. This field counts the number of times a buffer is successfully found in the cache but the contents are invalid.
F11BC$L_MISSES	Buffer not found. This field counts the number of times the buffer being sought is not present in the cache.

(continued on next page)

Table 4–1 (Cont.): Contents of the Cache Header

Field Name	Description
F11BC$L_DISK_READS	Number of read operations from disk into the buffer.
F11BC$L_DISK_WRITES	Number of write operations from the buffer to disk.
F11BC$L_CACHE_SERIAL	Number of cache serialization calls.
F11BC$L_CACHE_STALLS	Number of cache serialization stalls.
F11BC$L_BUFFER_STALLS	Number of stalls caused by the lack of available buffers.
F11BC$T_CACHENAME	Name of the cache. This field contains the device and cache name for which the cache was created. The format is **device:"xqpcache"**. The cache name tells whether the disk is using the default systemwide file system cache or whether the disk was mounted using a private cache. If the disk is using a systemwide cache, the cache name refers to the system disk because that disk is mounted first.

Figure 4–4 shows the format of the four word subfields of the F11BC$W_POOLCNT field.

Figure 4–4: Format of the Four Buffer Pool Subfields

ZK-9589-HC

4.2.3 Buffer Descriptors

A **buffer descriptor**, or BFRD, identifies the contents of a given buffer and its status. Among other things, a BFRD identifies the LBN and the UCB from which a buffer's contents were obtained. Each BFRD may be accessed quickly and efficiently by using the modulus of the LBN and the LBN hash count as an index into the LBN hash table.

Because the buffer descriptors, lock descriptors, and block buffers are all arrays, they are generally referenced by array index. The total count is limited to 65K, so the array index is limited to a word, saving space in the descriptors.

Each buffer in the buffer pools is represented by a buffer descriptor. Because of this one-to-one correspondence, the buffer descriptors, like the buffers themselves, are likewise divided into pools. The buffer and its descriptor are associated simply by using a common index value to locate either of them. The pool to which a BFRD belongs can be found in the BFRD$B_FLAGS field.

There are as many BFRDs as buffers, so the size of the descriptor area is directly proportional to the number of buffers in the cache.

Free BFRDs may be found on their respective LRU queues. If the BFRD is not in the LRU queue, it is on an in-process queue.

The fields of a buffer descriptor are shown in Figure 4–5 and are described in Table 4–2.

Figure 4–5: Format of a Buffer Descriptor

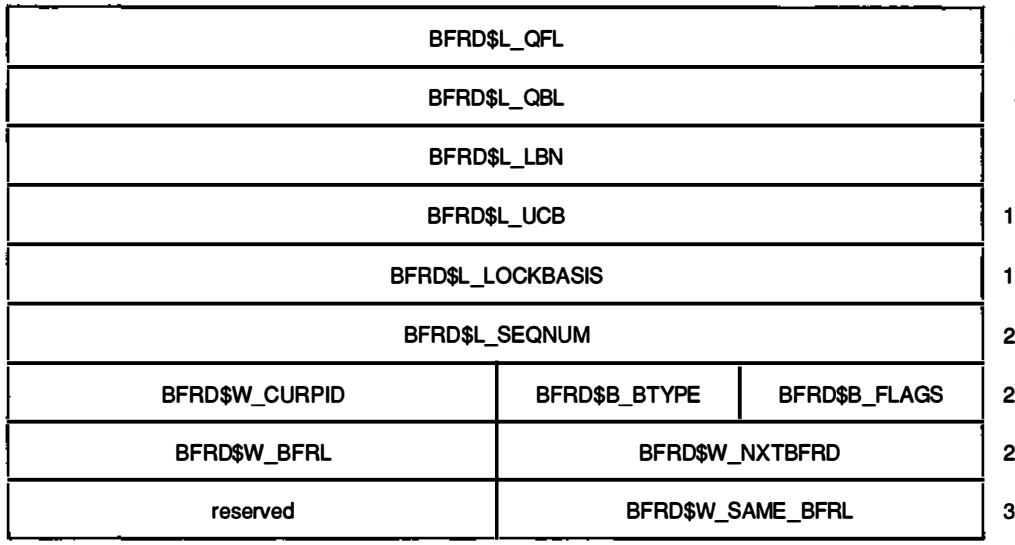


Table 4–2: Contents of a Buffer Descriptor

Field Name	Description
BFRD$L_QFL	Queue forward link. The queue can be either an in-process queue (BFR_LIST) or a pool queue (POOL_LRU).
BFRD$L_QBL	Queue backward link. The queue can be either an in-process queue (BFR_LIST) or a pool queue (POOL_LRU).
BFRD$L_LBN	LBN of buffer. This field, with the BFRD$L_UCB field, uniquely identifies the contents of the buffer.
BFRD$L_UCB	UCB of buffer. This field, with the BFRD$L_LBN field, uniquely identifies the contents of the buffer.
BFRD$L_LOCKBASIS	Unique file identifier. This field contains the FID (without its sequence number) used in the lock. This number follows the prefix F11B$s in the resource name.
BFRD$L_SEQNUM	Buffer validation sequence number. This field contains the clusterwide buffer sequence number initially obtained from the value block.
BFRD$B_FLAGS	Status flags. The following fields are defined within BFRD$B_FLAGS:
	BFRD$V_POOL — Pool number to which this buffer belongs.
	BFRD$V_DIRTY — Dirty buffer. If set, this bit indicates that the buffer has been modified.
	BFRD$V_VALID — Valid buffer. If set, this bit indicates that the buffer's contents are current and may be used as is.
BFRD$B_BTYPE	Buffer type. The following chart shows the possible buffer values and their meanings.

Value	Description
0	Header block
1	Bitmap block
2	Directory data block
3	Index file block
4	Random data block
5	Quota file block
6	Directory index block

(continued on next page)

Table 4–2 (Cont.): Contents of a Buffer Descriptor

Field Name	Description
BFRD$W_CURPID	Process index of the current process. This field contains the process index of the process that owns the buffer.
BFRD$W_NXTBFRD	Index of next BFRD. This field contains the index of the next buffer descriptor in the hash chain.
BFRD$W_BFRL	Index to buffer lock hash chain. This field contains the index to the BFRL to which this buffer belongs.
BFRD$W_SAME_BFRL	Index to the next BFRD under the same BFRL.

4.2.4 Buffer Lock Block Descriptors

A **buffer lock block descriptor,** or BFRL, describes the lock associated with each buffer in the cache. Because each buffer descriptor may have a lock associated with it, the number of BFRLs equals the number of BFRDs. Unlike buffer descriptors, however, buffer lock descriptors are not divided into pools.

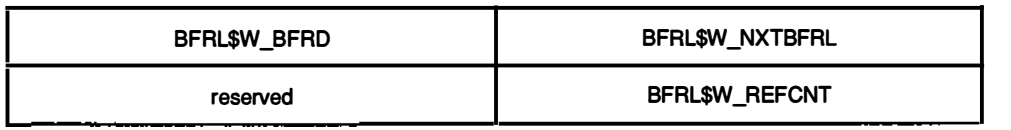
A single lock may be associated with more than one buffer descriptor when multiple blocks are read under the same lock, such as for the quota file, a directory, or a multiheader file.

Essentially, BFRLs are used to associate buffers with a particular file. One BFRL exists for each BFRD for a file, and one BFRD represents one disk block. Thus, multiple BFRDs per BFRL are possible, but multiple BFRLs per BFRD are not.

BFRLs are used to keep track of system-owned null locks that are always on a block in one of the block caches. They are not used for protected write serialization process locks. For more information on serialization of file system activity on a single node and in a VAXcluster environment, see Chapters 7 and 8.

The fields of the buffer lock block descriptor are shown in Figure 4–6 and are described in Table 4–3.

Figure 4–6: Format of the Buffer Lock Block Descriptor



(continued on next page)

Figure 4–6 (Cont.): Format of the Buffer Lock Block Descriptor

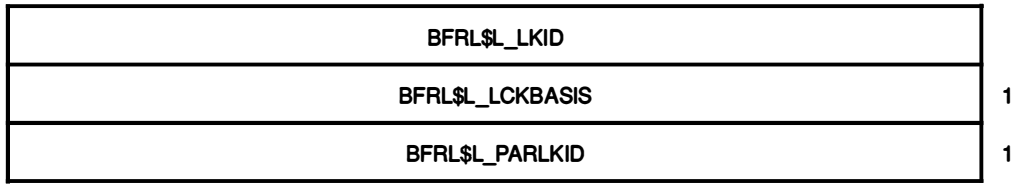
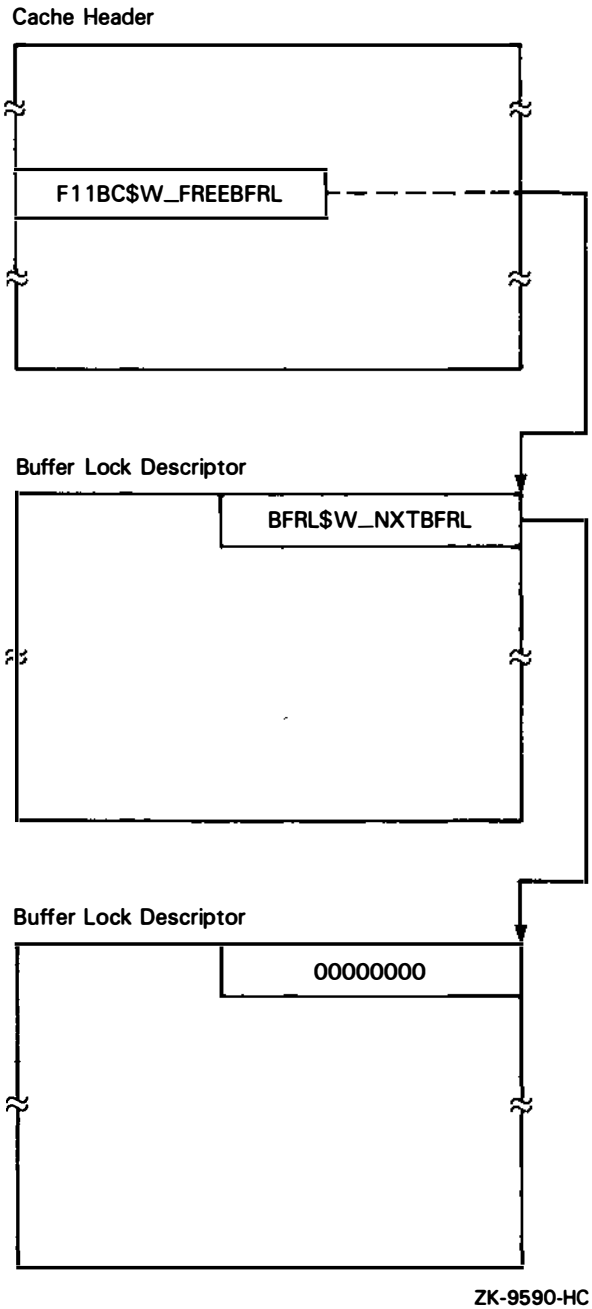


Table 4–3: Contents of the Buffer Lock Block Descriptor

Field Name	Description
BFRL$W_NXTBFRL	Index to the next BFRL in the hash chain. This field also serves as an index to the next entry in the buffer lock descriptor chain (F11BC$W_FREEBFRL) that contains all the BFRLs that are not in use. The buffer index is used as a word pointer. The last entry in the list is indicated by a value of 0. The most-recently-used entry is inserted at the beginning of the list, and all allocations also take place from the beginning of the list. The free BFRL list is shown in Figure 4–7.
BFRL$W_BFRD	Index to first BFRD.
BFRL$W_REFCNT	Number of buffers represented by this lock.
BFRL$L_LKID	Lock ID of buffer lock.
BFRL$L_LCKBASIS	Unique file identifier. This FID follows the prefix F11B$s to form the resource name. Because this field is used as an index to the lock hash table, it provides fast access to the locks.
BFRL$L_PARLKID	Parent lock ID. This field contains the lock ID of the volume allocation lock of the volume to which the buffer contents belong.

Figure 4–7: Free Buffer Lock Descriptor List



4.2.5 LBN and the Lock Basis Hash Tables

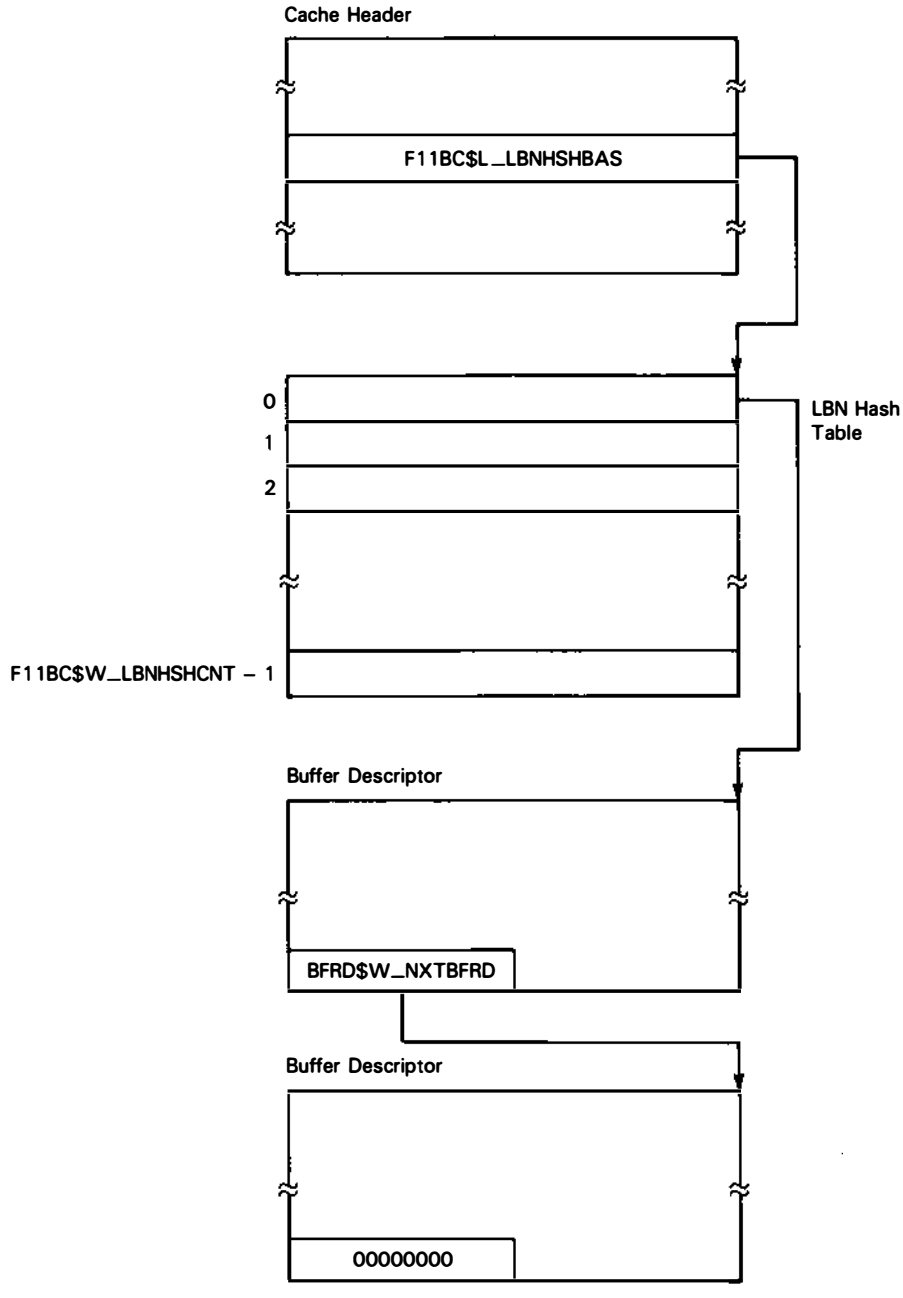
The **LBN hash table** and the **lock basis hash table** (or buffer lock hash table) both contain a minimum of one word each per buffer. An extra page is added to the total so the buffers themselves are always aligned on a page boundary regardless of where the space is actually allocated in paged pool. Any extra space between the lock descriptors and the start of the buffers is split between the two hash tables.

The chief purpose of the LBN hash table is to provide fast access to the LBNs in the cache. The hash function is a modulo function using the desired LBN and the size of the hash table (contained in F11BC$W_LBNHSHCNT) in words. Collisions are handled by chaining the BFRDs. The last entry in the hash chain is indicated by a value of 0. The hash table entries and chain links are both buffer index word pointers.

Buffers without valid blocks are not contained in the hash table.

Figure 4–8 shows the relationship between the cache header, the LBN hash table, and the buffer descriptors.

Figure 4–8: Layout of the LBN Hash Table



The lock basis hash table also provides quick access to buffer locks by using a lock basis, which is a longword composed of a relative volume number, an extended file number, and a file ID. This lock basis is used in the lock resource name, prefixed by **F11B$s.** The BFRL$L_PARLKID field identifies the volume on which the file is located.

The hash function is a modulo function using a unique identifier (obtained by adding the BFRL$L_LCKBASIS and the BFRL$L_PARLKID fields) and the size of the hash table (contained in F11BC$W_BLHSHCNT) in words. The last entry in the hash chain is indicated by a value of 0. The hash table entries and chain links are both buffer index word pointers.

Figure 4–9 shows the relationship between the cache header, the lock basis hash table, and the buffer lock descriptors.

Figure 4–9: Layout of the Lock Basis Hash Table

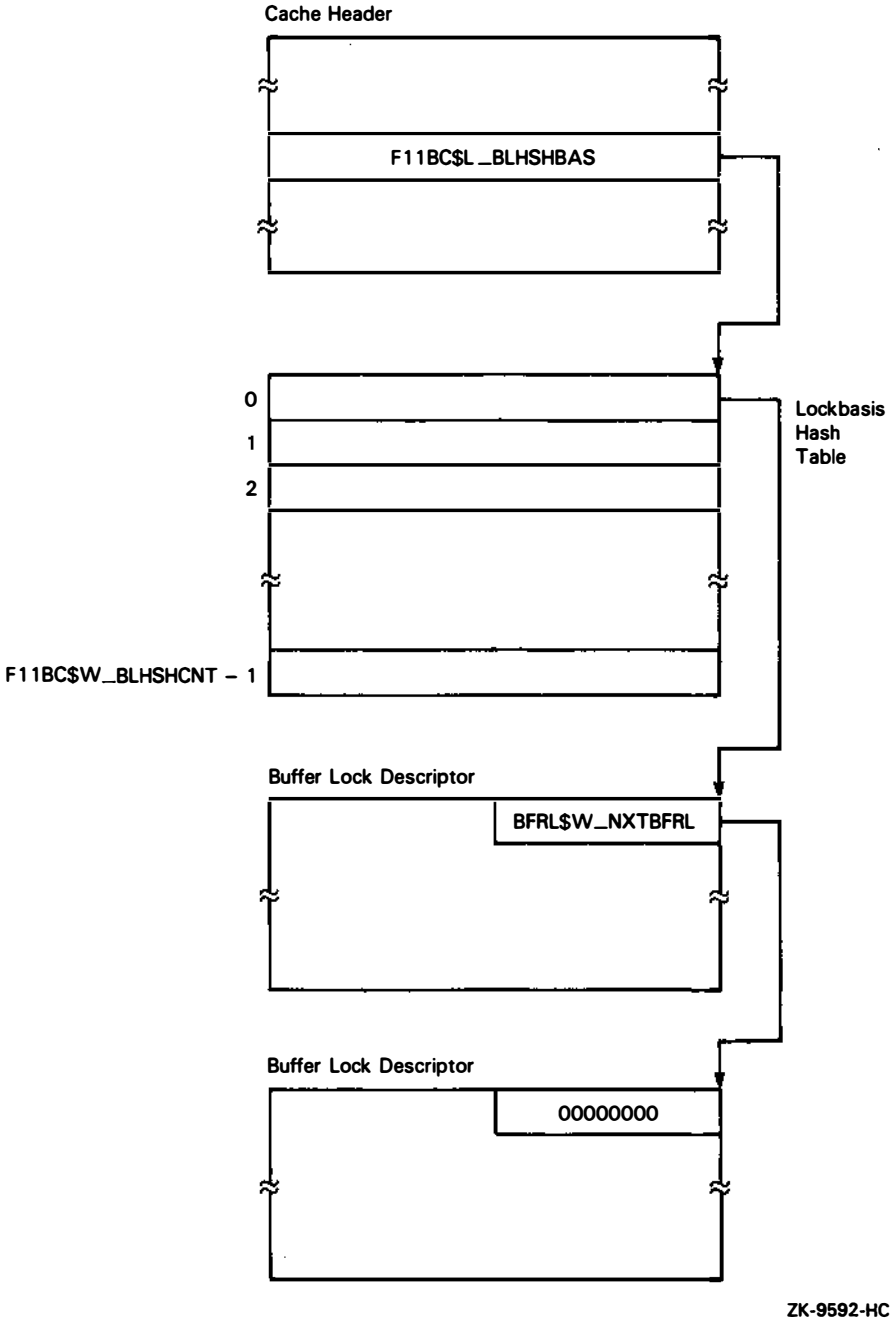
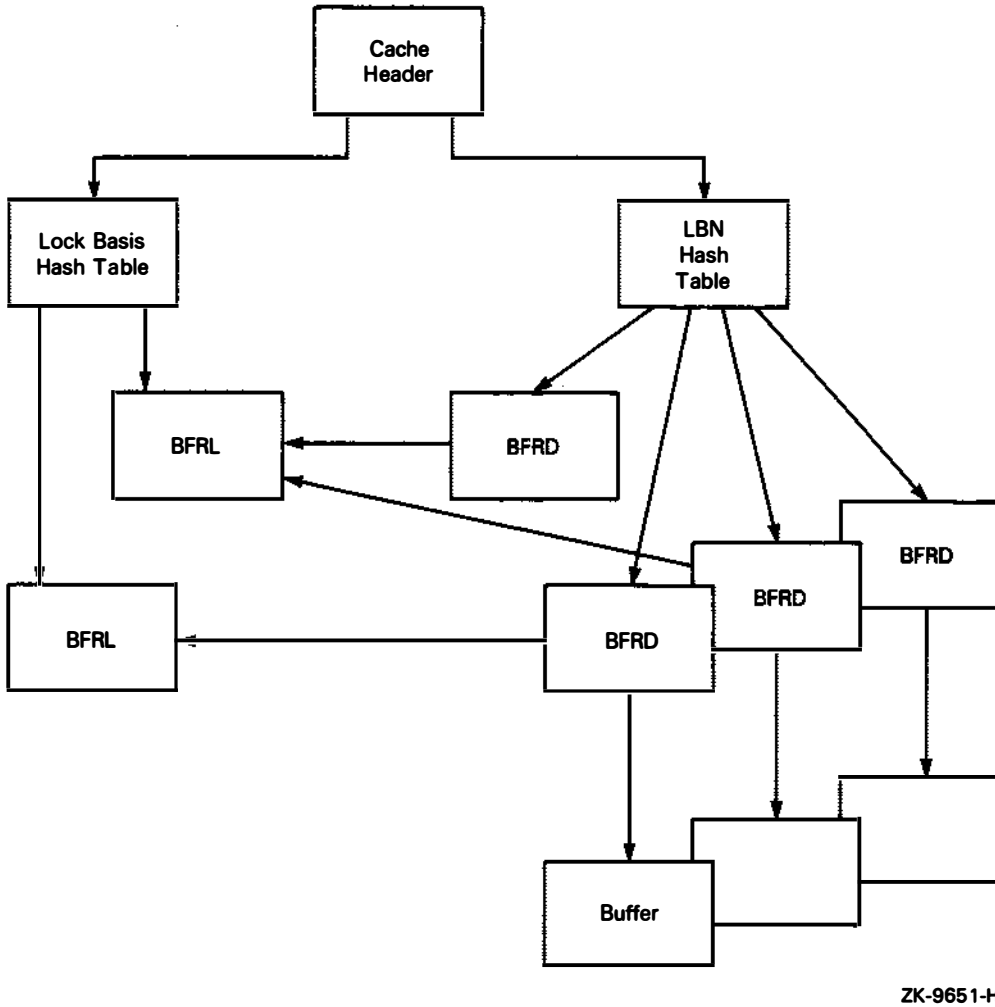


Figure 4–10 shows the relationship between the cache header, the buffer descriptors, and the buffer pool.

Figure 4–10: Layout of the XQP Block Cache



4.2.6 Buffer Pools

The remainder of the I/O buffer cache is divided into four pools of data buffers:

- Storage bitmap blocks and the SCB

  These are all the data blocks mapped by the BITMAP.SYS file. See Section 4.2.6.1.

- Directory data blocks, random data blocks, and quota file blocks

  This is the only pool on which multiblock read operations may be performed. See Sections 4.2.6.3 and 4.2.10.

- File headers and index file bitmap blocks

  These are all data blocks mapped by the index file. See Section 4.2.6.2.

- Directory index blocks

  This cache is used by the directory index caching mechanism. These pages are not I/O buffers, but they are managed by the buffer caching routines because they provide the necessary cluster validation. See Section 4.2.6.4.

The constant F11BC$K_NUM_POOLS, currently equal to 4, indicates the number of buffer pools.

A least recently used, or LRU, algorithm is used to replace buffers (that is, the first three cache pools). When the desired disk block cannot be found in the cache, the buffer that was referenced earliest is discarded and replaced with the desired block.

This algorithm is accomplished by linking all BFRDs for a given pool onto a queue header for that pool. This address of this queue header is contained in the F11BC$Q_POOL_LRU field, which is actually a vector of four queue headers that represent the four pools.

The pool to which a buffer belongs is contained in the BFRD$B_FLAGS field. Also, because more than one buffer type resides in a pool, the BFRD$B_BTYPE field is used to differentiate between the buffer types.

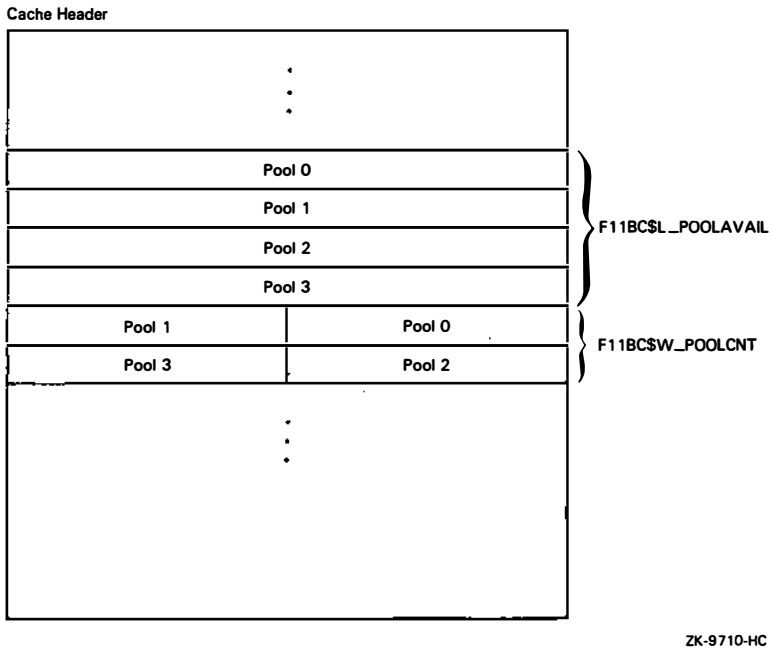
The four pools are numbered from 0 through 3, and the POOL array is used to find the pools.

Buffers read by file system operations are not "locked down"; rather, they may be reused any time further reads are issued in the same buffer pool. The buffer manager guarantees that the last n buffers read in each pool will be available, where n is the minimum buffer credit reserved for that pool (see Section 4.3). If it is necessary to read a set of blocks that exceeds the minimum credit, the local and global variables pointing to the blocks that were read earlier may no longer be valid. In this case, the original blocks must be read again.

When there is no file system activity, the four values in the F11BC$L_POOLAVAIL vector equal the four values in the F11BC$W_POOLCNT field. In addition, all buffers for a given pool are linked onto their respective F11BC$Q_POOL_LRU queue headers.

Figure 4–11 shows the location of the available queue and the POOLCNT array in the cache header.

Figure 4–11: Relationship Between F11BC$L_POOLAVAIL and F11BC$W_POOLCNT

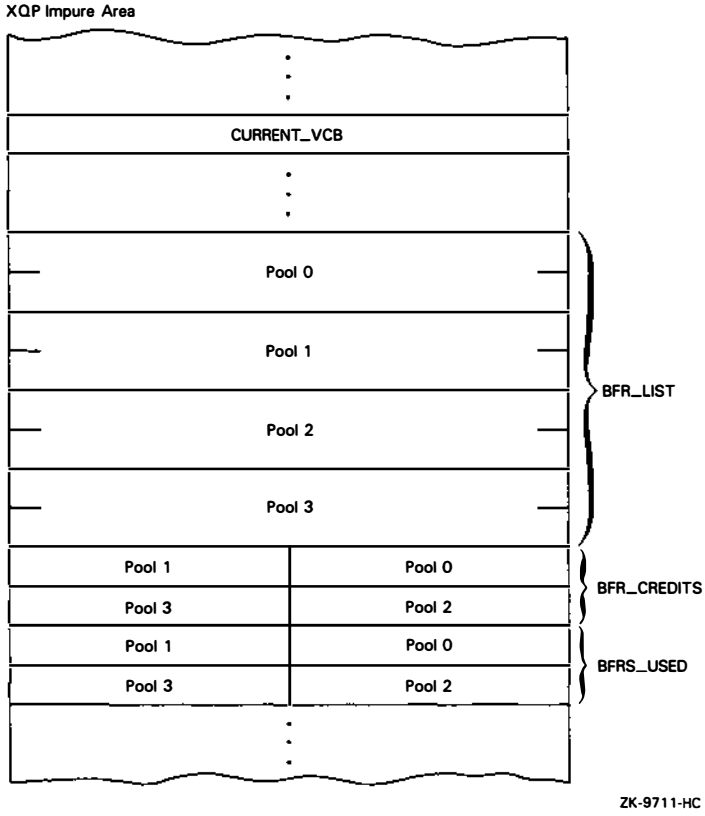


When a buffer is being used by a particular process during an operation, it is removed from the POOL_LRU queue and inserted onto a per-process (or in-process) BFR_LIST queue. The BFR_LIST structure itself is a vector of queue headers, one for each pool. Each process also has two four-element vectors,

BFR_CREDITS and BFRS_USED, representing, respectively, the number of buffers reserved and the number actually in use.

Figure 4–12 shows the in-process queue and two vectors in the XQP impure area that keep track of buffer information.

Figure 4–12: Location of BFR_LIST, BFR_CREDITS, and BFRS_USED



The purpose behind buffer credits is to avoid resource deadlocks. The credits prevent stalls in critical sections of XQP code (any section where a lock is held on a systemwide resource such as a cache). A stall could potentially cause a deadlock. If credits are obtained before any processing can proceed, XQP code can be executed more quickly and without stalling. For more information on how buffer credits are extended, refer to Section 4.3.1.

The number of BFRDs in each queue header in the BFR_LIST structure must always correspond to the value in the BFRS_USED vector. When a BFRD is on the BFR_LIST queue, the BFRD$W_CURPID field contains the internal PID index for that process.

4.2.6.1 Storage Bitmap Cache

The **storage bitmap cache** contains blocks from the storage bitmaps of volumes mounted on the system. Each bit in the bitmap for a given volume represents one disk cluster of disk space.

The size of this cache is controlled by the ACP_MAPCACHE system parameter. The default value is eight blocks, which is sufficient unless many volumes are mounted and they all have a significant amount of file creation and extension activity.

Caching the entire bitmap on a volume that is heavily fragmented may benefit performance; the file system makes multiple passes over the bitmap in the following cases:

- When trying to cache a reasonable amount of disk space in the extent cache
- When determining whether or not a contiguous file creation or extension operation may be completed if the existing entries in the bitmap cache cannot satisfy the space requirement

The file system makes three complete searches of the storage bitmap to satisfy a contiguous-best-try allocation request. If this request cannot be satisfied, the allocation will be fragmented.

Access to this cache is synchronized by the volume lock of the block being accessed.

4.2.6.2 File Header and Index File Bitmap Cache

The **file header cache** greatly decreases the time needed to open a file. The file header contains information used to construct the FCB, the data structure that the file system uses to control access to the file. It also contains the mapping pointers that define the locations of all the data blocks associated with the file.

This pool also contains the **index file bitmap cache**. This cache holds blocks from the index file bitmap that are used to allocate unused file IDs to the FID cache when the FID cache is depleted.

The size of this cache is controlled by the ACP_HDRCACHE system parameter. The default value of 128 pages allows slightly less than 128 file headers to be cached. Files that have multiple extension headers because of fragmentation and large ACLs take up more space in the cache and decrease the number of files for which headers are cached.

If the file header is in the cache, the performance decrease caused by a window turn is lessened because the file header (or headers) contains all the mapping information for the file.

Each entry (that is, each page) in this cache has its own lock.

4.2.6.3 Directory Data Block Cache

The **directory data block cache** contains directory data blocks so that file lookups can be performed more quickly. That is, the cache contains the contents of recently referenced directories.

The size of this cache is controlled by the ACP_DIRCACHE system parameter, and the default is 80 pages. The ACP_MAXREAD system parameter controls the maximum number of directory data blocks read in one I/O operation. For more information, refer to Section 4.4.

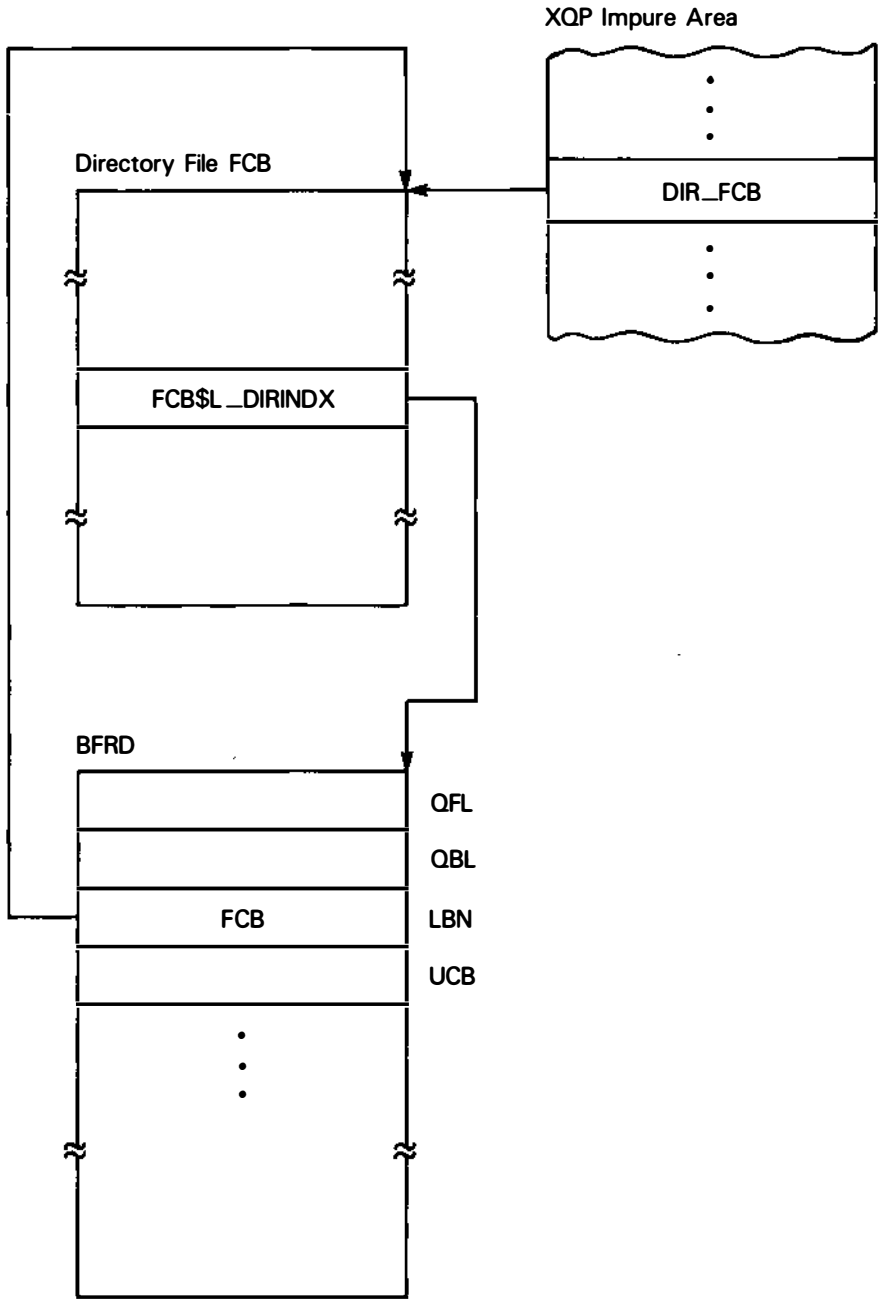
Each entry (that is, each page) in this cache is synchronized by its own lock.

4.2.6.4 Directory Index Cache

The **directory index cache** is the fourth pool in the buffer cache. It is located in paged pool. Unlike the other pools, this pool does not contain buffers; rather, it contains an index into a given directory file, constructed while the directory is being processed.

Figure 4–13 shows the relationship between the directory file FCB, the XQP impure area, and the buffer descriptor.

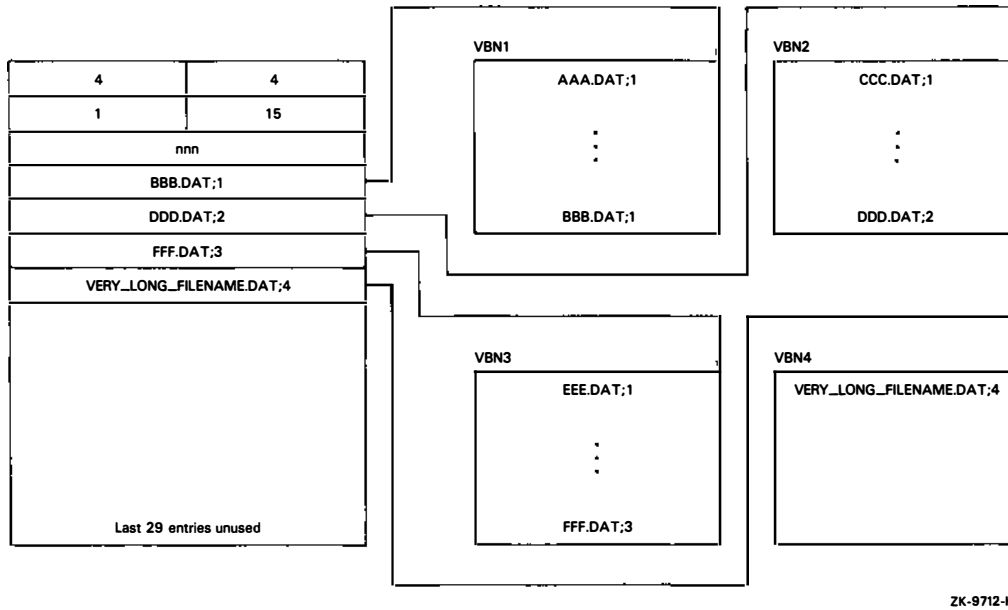
Figure 4–13: Locating the Directory Index Cache



Each entry in the DINDX cache points to a particular location in a directory file. One page is dedicated to each directory; entries consist of the last file entered in each directory data block. This cache allows the file system to search a directory for a given file without having to do a linear search on the entire directory. This advantage saves time and also reduces the number of block buffers used in the directory cache.

Figure 4–14 shows an example of entries in the directory index cache.

Figure 4–14: Directory Index Cache Entries



The size of this cache is controlled by the ACP_DINDXCACHE system parameter. Its value should equal the number of active directories on the system. For large directories, each cache entry represents a group of blocks rather than a single block, so that the number of groups does not exceed the fixed size of the index.

For more information on the directory index cache, see Section 8.6.5.

4.2.7 Specialized Caches

There are three specialized caches on a disk volume:

- Extent cache
- File ID cache
- Quota cache

The **volume cache block** (VCA) points to two of these caches: the file ID cache and the extent cache. These two caches are located together in one block. The quota cache, however, is located separately in another block.

The volume cache block is located in nonpaged pool, and is pointed to by the VCB.

The fields of the volume cache block are shown in Figure 4–15 and are described in Table 4–4.

Figure 4–15: Format of the Volume Cache Block

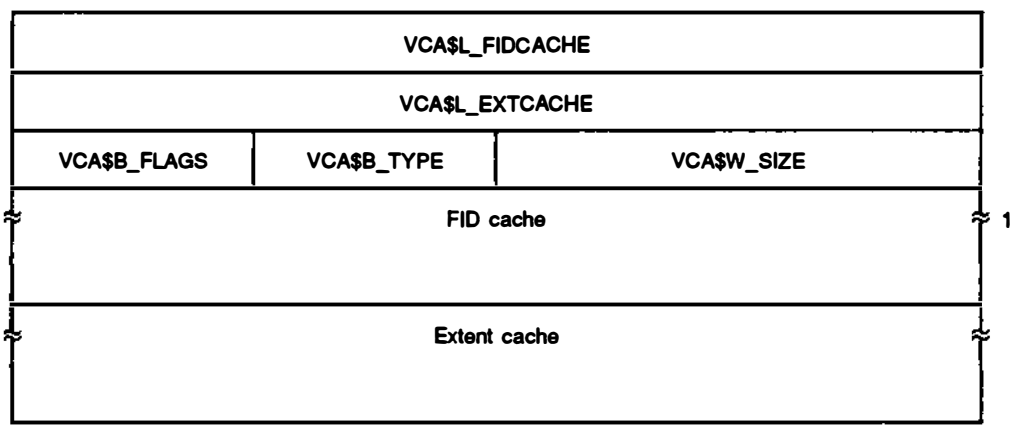


Table 4–4: Contents of the Volume Cache Block

Field Name	Description	
VCA$L_FIDCACHE	Pointer to the file ID cache.	
VCA$L_EXTCACHE	Pointer to the extent cache.	
VCA$W_SIZE	Block size.	
VCA$B_TYPE	Block type code.	
VCA$B_FLAGS	Flags. The following fields are defined within VCA$B_FLAGS:	
	VCA$V_FIDC_VALID	FID cache valid
	VCA$V_EXTC_VALID	Extent cache valid
	VCA$V_FIDC_FLUSH	FID cache to be flushed to disk
	VCA$V_EXTC_FLUSH	Extent cache to be flushed to disk

4.2.8 Extent Cache

The **extent cache** is essentially a preallocated section of the storage bitmap file. It contains a list of known free (unused) extents that can be allocated when a file is newly created or extended. An extent is a pointer (consisting of an LBN and a size) that maps a logically contiguous area of disk space on a disk volume.

The main purpose of the extent cache is to maintain a certain fraction of the free space on the disk in the extent cache. It is found from the addresses in the VCB$L_CACHE and the VCA$L_EXTCACHE fields. A default value of 64 extents is controlled by the system parameter ACP_EXTCACHE. The cache is allocated from nonpaged pool.

When the file system tries to allocate an extent, the extent cache is checked first. If the cache allocation fails, the storage bitmap itself is scanned. It is scanned twice: once from the given starting point to the end, and, if necessary, again from the beginning of the bitmap to the end.

After the allocation, an attempt is made to refill the extent cache from the bitmap block in memory, and then from the bitmap itself. The VBN of the bitmap block last used to allocate disk blocks is recorded in the storage bitmap VBN field (SBMAPVBN) in the allocation lock value block.

When the file system returns blocks (for example, when a file is deleted), they are returned to the extent cache. If the cache overflows, some extents are purged back to the storage map.

Note that the system parameter ACP_EXTLIMIT controls only how many extents are actually in the cache, and this number will never be more than the following value:

$$\frac{ACP\_EXTLIMIT}{10} \textit{ percent of the free space on the volume}$$

Once this limit is reached, no more extents will be added to the cache. On volumes with large amounts of contiguous free space, the cache will not contain many extents.

The extent cache consists of the cache header, followed by a quadword vector of extents, densely packed. Each quadword contains a block count and a starting LBN.

The fields of the extent cache are shown in Figure 4–16 and are described in Table 4–5.

Figure 4–16: Format of the Extent Cache Block

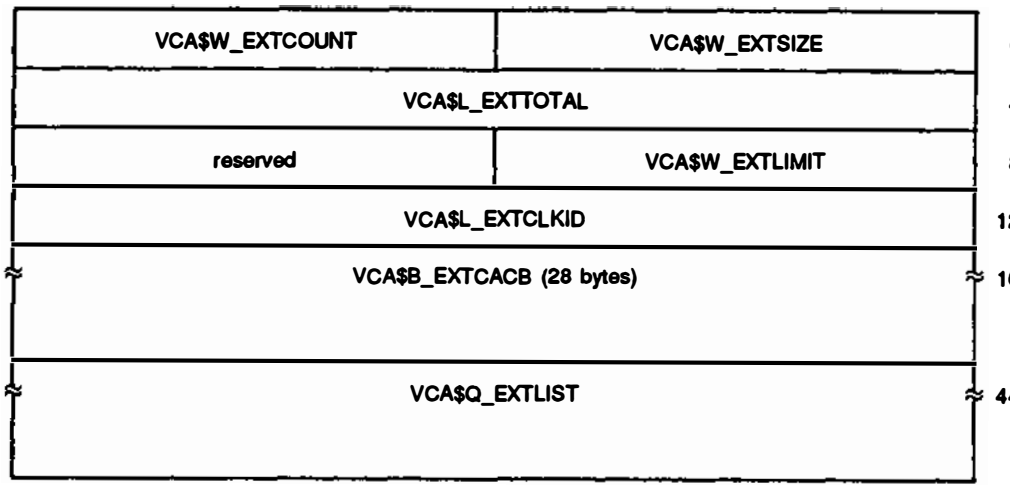


Table 4–5: Contents of the Extent Cache Block

Field Name	Description
VCA$W_EXTSIZE	Number of entries allocated, controlled by the system parameter ACP_EXTLIMIT
VCA$W_EXTCOUNT	Number of entries currently in use
VCA$L_EXTTOTAL	Total number of blocks contained in cache
VCA$W_EXTLIMIT	Limit of volume to be cached, in tenths of a percent
VCA$L_EXTCLKID	Extent cache lock ID
VCA$B_EXTCACB	Extent cache blocking AST control block (ACB)
VCA$Q_EXTLIST	First entry in list

Figure 4–17 shows the format of an extent cache entry. The VCA$L_EXTBLOCKS field represents the number of blocks, and the VCA$L_EXTLBN field contains the starting LBN.

Figure 4–17: Format of an Extent Cache Entry

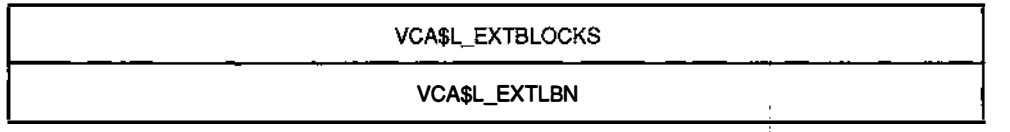
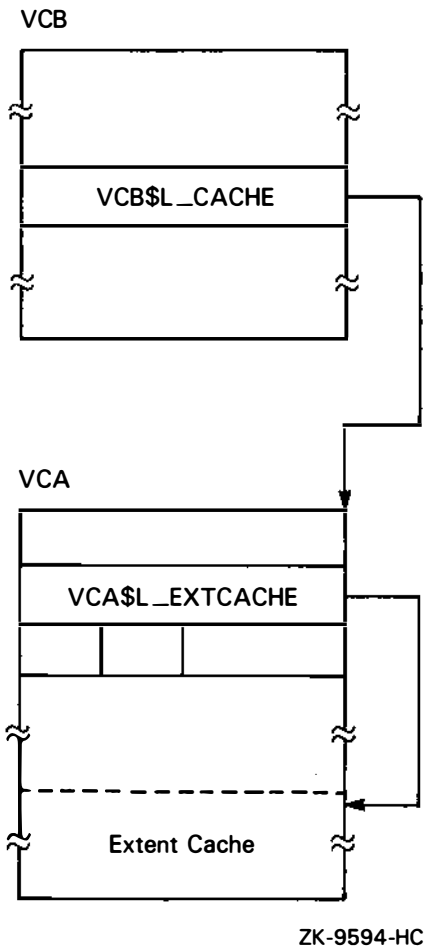


Figure 4–18 shows how the address in the VCB$L_CACHE field locates the volume cache block and the extent cache.

Figure 4–18: Locating the Extent Cache from the Volume Cache Block



4.2.9 File ID Cache

The **file ID**, or **FID cache**, is effectively a preallocated section of the index file bitmap. In other words, it is a cache of unused file identifiers that can be allocated when a file is created or an extension header for a file is needed.

The FID cache is found using the addresses in the VCB$L_CACHE and the VCA$L_FIDCACHE fields. The default value of 64 file IDs is obtained from the system parameter ACP_FIDCACHE. The cache is allocated from nonpaged pool.

When the file system allocates a file ID, the FID cache is checked first. If the cache is empty, blocks are read from the index file map until a block with a free bit is found, and the free bits are then added to the FID cache. The file header for this file ID must, of course, be available.

The index file bitmap block is written to the disk during this operation. If it is not written back, two processes searching for a free FID in the bitmap might allocate the same FID.

The fields of the file ID cache are shown in Figure 4–19 and are described in Table 4–6.

Figure 4–19: Format of the File ID Cache Block

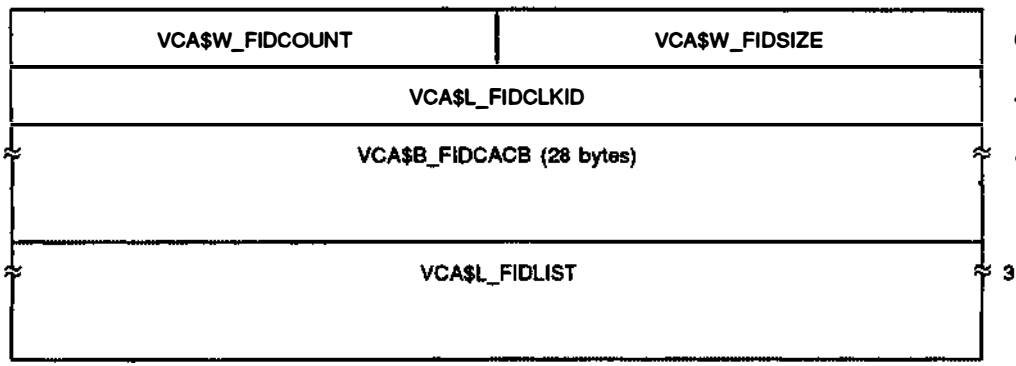
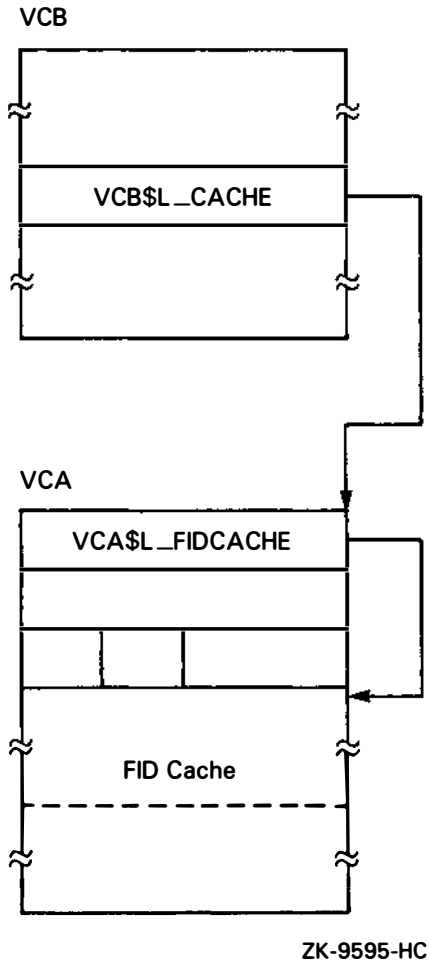


Table 4–6: Contents of the File ID Cache Block

Field Name	Description
VCA$W_FIDSIZE	Number of entries allocated to the cache
VCA$W_FIDCOUNT	Number of entries present in the cache
VCA$L_FIDCLKID	FID cache lock ID
VCA$B_FIDCACB	FID cache blocking AST control block
VCA$L_FIDLIST	First entry in list

Figure 4–20 shows how the address contained in the VCB$L_CACHE field locates the volume cache block and the file ID cache.

Figure 4–20: Locating the File ID Cache from the Volume Cache Block



4.2.10 Quota Cache

The **quota cache** contains UIC-based entries to keep track of the allowed usage and current usage of disk blocks without continually having to read and write the blocks of the QUOTA.SYS file. This cache is irrelevant if the volume has been mounted with the /NOQUOTA qualifier or does not have quotas enabled. The quota cache is pointed to by the address in the VCB$L_QUOCACHE field. It is allocated from nonpaged pool.

Each quota entry contains a UIC, the quota information, a lock status block used with the quota cache entry locks, the quota file record number, and LRU indexes. The cache header contains a LRU counter; when a new entry is added, the value is put into the entry, and this counter is incremented.

When the cache is flushed, each entry is returned to disk, and the corresponding record on disk is located and updated. Any quota entry locks are released, and the quota cache lock is converted to null mode.

The quota cache consists of the cache header, followed by the cache entries. Each cache entry is a block as defined below.

The fields of the quota cache header block are shown in Figure 4–21 and are described in Table 4–7.

Figure 4–21: Format of the Quota Cache Header Block

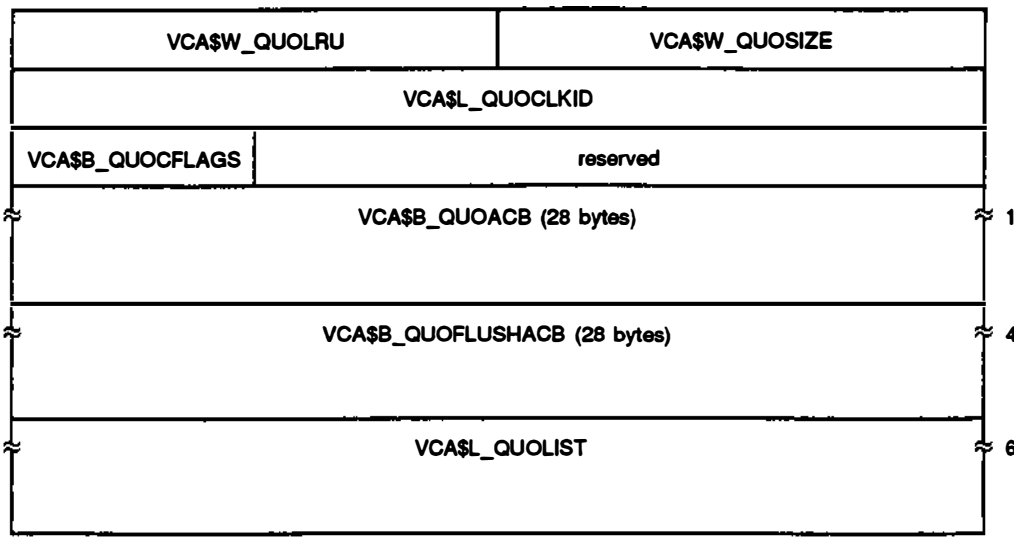


Table 4–7: Contents of the Quota Cache Header Block

Field Name	Description	
VCA$W_QUOSIZE	Number of entries allocated.	
VCA$W_QUOLRU	Current LRU counter.	
VCA$L_QUOCLKID	Whole cache lock ID.	
VCA$B_QUOCFLAGS	Flags. The following fields are defined within VCA$B_QUOCFLAGS:	
	VCA$V_CACHEVALID	Valid cache. The cache is enabled and may contain entries.
	VCA$V_CACHEFLUSH	Cache is to be flushed.
VCA$B_QUOACB	ACB to deliver blocking AST on the cache lock.	
VCA$B_QUOFLUSHACB	ACB to deliver cache flush AST.	
VCA$L_QUOLIST	Start of entries.	

Figure 4–22 shows the fields of a quota cache entry. These 24 bytes form the substructure VCA$R_QUOLOCK, which is used for the lock status block. Note that the fields of the figure run right to left.

Figure 4–22: Format of a Quota Cache Entry

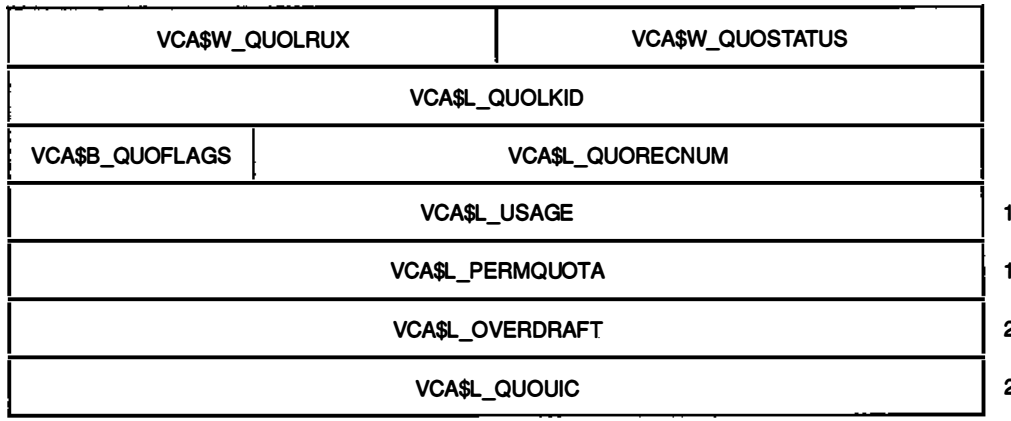
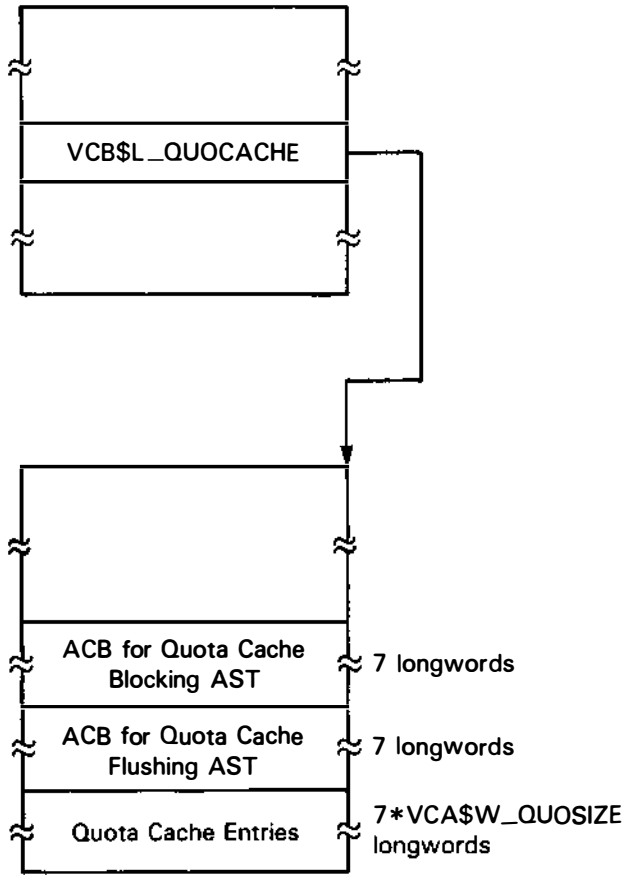


Table 4–8: Contents of a Quota Cache Entry

Field Name	Description	
VCA$W_QUOSTATUS	Current $ENQ status. This field is also used to hold the value of VCA$W_QUOINDEX, which represents the number of this entry in the quota cache.	
VCA$W_QUOLRUX	LRU index for entry.	
VCA$L_QUOLKID	Lock ID of cache entry.	
VCA$L_QUORECNUM	Record number.	
VCA$B_QUOFLAGS	Flags. The following fields are defined within VCA$B_QUOFLAGS:	
	VCA$V_QUOVALID	Valid entry is present. This flag indicates that the quota cache entry is valid and contains current data.
	VCA$V_QUODIRTY	Modified entry is present. This flag indicates that the quota cache entry has been modified but not yet written to the quota file.
VCA$L_USAGE	Current usage.	
VCA$L_PERMQUOTA	Permanent quota.	
VCA$L_OVERDRAFT	Overdraft limit.	
VCA$L_QUOUIC	UIC.	

Figure 4–23 shows how the quota cache may be located from the VCB.

Figure 4–23: Locating the Quota Cache from the Volume Control Block



4.3 Obtaining Buffers

Before a given file system operation is allowed to use any buffers in the cache or before any operation is allowed to hold any locks, the minimum number of buffers required to perform the operation must first be reserved. The process of obtaining buffers is managed by maintaining counters, one for each pool, in the fixed overhead area. These counters represent the number of buffers currently reserved for concurrent file system activity. The number of buffers currently available in each pool is contained in the F11BC$L_POOLAVAIL vector. The currently required buffer credits are as follows:

- One bitmap block buffer
- Two directory data block buffers
- Three file header buffers
- One directory index buffer

A process is stalled until enough buffers are available to complete the activity. The F11BC$Q_POOL_WAITQ vector has listheads for each pool, and an IRP is queued on the pool wait queue while the request is stalled. The process is sent an AST when buffer credits are returned.

The logic behind the process of obtaining buffers prevents resource deadlocks that could result if the following three conditions exist:

- A partially completed operation requires additional buffers to complete.
- No free buffers are available.
- Other processes holding buffers are waiting for the partially completed operation to complete.

The obtaining of credits is controlled by the cache interlock. The buffer credits are returned to the free pool counts, under the cache interlock, but only if the buffers are not in use. If there is a process either on the pool wait queue or on the ambiguity queue, the process is added to the cache interlock queue after the current process. The process is awakened when the cache interlock is released. For more information on the cache interlock, see Chapter 7.

4.3.1 Extending Buffer Credits

A certain number of buffers must be reserved before any process is allowed to start a file system operation. For example, three buffers from the file header pool are always reserved. If only six buffers were resident in the file header pool (set by the ACP_HDRCACHE system parameter), only two processes would be allowed to proceed concurrently. Until one of them completed, a third process could be stalled.

In this situation, if a file with four headers is accessed, the process would have to discard from its buffer the first header read from its BFR_LIST and use that buffer to read the fourth header. The BFR_LIST structure is managed with an LRU algorithm, and the oldest buffer is always discarded.

However, if there are more than six unreserved buffers in a given pool, additional buffer credits are extended to a process to avoid invalidating a recently accessed buffer (as the previous example did). This operation is done by decrementing the contents of the F11BC$L_POOLAVAIL field and then incrementing a pool counter when the additional buffer is desired, but only if the number in the F11BC$L_POOLAVAIL field is greater than or equal to six.

In other words, the file system uses as many buffers from the pool as are available without affecting other users.

4.4 Multiblock Disk Read Operations

The directory and quota file data block pool allow multiblock read operations. A contiguous group of buffers is assembled to be used in a single multiblock QIO operation when the buffer desired by the caller is not already in the cache. Directory and quota file processing also request multiblock read operations. The number of buffers assembed in the caches is limited by the value of the system parameter ACP_MAXREAD.

The contiguous group of buffers is assembled starting with a BFRD pulled from the POOL_LRU list. The file system tries to assemble adjacent BFRDs in ascending memory sequence. If the end of the pool is reached, the file system then tries to proceed from the starting BFRD in descending memory sequence.

Assembling a group of contiguous buffers may fail under three conditions:

- If any BFRD is already in use (that is, the BFRD$W_CURPID is nonzero)
- If the LBN the file system intended to read into that BFRD is already in the cache somewhere
- If the buffer credits for the process are exceeded and the process is not extended any more credits

4.5 Disk Write Operations

All writing to disk (except for normal virtual write functions and erase functions) is performed by the WRITE_BLOCK routine (in RDBLOK) or the WRITE_HEADER routine (also in RDBLOK), which performs a checksum first. Buffers can be explicitly written in this way.

WRITE_BLOCK is invoked automatically when it is necessary to remove a buffer from the in-process list (dirty buffers must be only on the in-process list). WRITE_DIRTY can be called to write all dirty buffers associated with a lock basis (which implies that all buffers should be written).

TOSS_CACHE_DATA will do the same for a given lock array index, except that it also invalidates the cache buffers. This is done when closing a file opened using OPEN_FILE.

Most operations that modify buffers will simply mark them as dirty and allow CLEANUP to write them with the WRITE_DIRTY routine. There are various exceptions, as follows:

- ERR_CLEANUP force writes the current directory buffer when it performs a re-enter function.
- CREATE_HEADER force writes the index file header when advancing the EOF (not currently done). CREATE_HEADER also force writes blocks of the index file bitmap when filling the FID cache. DELETE_FID performs likewise when returning FIDs to the index file bitmap.
- DEALLOCATE_BAD force writes modified file headers. SCAN_BADLOG force writes the pending bad block file (BADLOG) file header when extending its header.
- MARK_DELETE force writes the updated (marked as deleted or actually deleted) headers out to disk. DELETE_FILE does likewise.
- EXTEND_CONTIG force writes data blocks as it copies them to the new extended contiguous file. The new header is force written. Likewise, SHUFFLE_DIR force writes directory blocks during its copy.
- TRUNCATE force writes the file header with the map pointers truncated so that the header is guaranteed to be updated before the storage bitmap shows the blocks as free.
- WRITE_AUDIT force writes all modified buffers given the primary lock basis before doing the FID_TO_SPEC translation, which releases the lock basis.

4.6 Systemwide Buffer Validation

Buffers are located in one of two places:

- In the system list, possibly marked as containing valid data read from disk
- In an in-process list, containing either valid data from disk or "dirty" data that has been modified but not yet written to disk

When a buffer is moved from the system list to the in-process list, it is read if the buffer descriptor describes it as invalid.

CREATE_BLOCK is a general-purpose routine that creates a new buffer filled with 0s. Under some circumstances, though, it is called with a backing LBN of –1. This means that the desired buffer is a scratch buffer and does not represent any data on disk.

The file system uses this technique in a truncate function. Two copies of the file header are temporarily needed because the updated file header must be written before the blocks are freed.

A truncate function performs the following steps when it is creating a new file header:

1. Calls CREATE_BLOCK with an LBN argument of –1
2. Copies the file header into the created block
3. Zeroes the map area of the original file header and updates it
4. Uses the file header copy to free the blocks
5. Deallocates the copy by calling INVALIDATE

4.6.1 Invalidating a Buffer

When a buffer is invalidated, it is moved to the front of the in-process LRU list and marked as not valid (but not dirty). The following operations will mark a buffer invalid:

- Any process that performs either a read or a write operation to the SCB (such as a dismount function) invalidates the buffers to ensure that the SCB is not cached on behalf of a shadow set (since mount verification writes asynchronously to the SCB).
- When a new header is being created, the file system may invalidate the header if it finds it does not want to use the header (if, for instance, the header appears to be valid). This invalidation prevents confusion if the header is found in the cache when it should not be.
- When a new header is being read and the operation fails, a test is performed to see if the block can be written. If the write operation succeeds, the buffer is invalidated, and a read operation is tried again.
- When an unused header is sought and found, the file system reads and validates it. However, the validation is expected to fail because only invalid headers should exist in the buffer. If the validation succeeds, then a problem may occur because the header already points to another file. In this case, the buffer is marked invalid and then discarded.
- Any buffer being discarded is also invalidated.

- When a directory is marked to be deleted, the first data block of the directory is read to make sure the directory is empty. The buffer read is invalidated since it is no longer needed. The directory cache is also flushed to write out its data blocks. Turning off the directory bit for a directory also flushes the directory blocks.
- When the volume is being dismounted or is being mounted with the /NOCACHE qualifier, the buffers of the cache are flushed. This operation invalidates all the buffers associated with a particular UCB. Buffers in the system list are purged; buffers in the process list are marked invalid; and buffers in other process lists are ignored.
- When a directory is write-accessed, the directory data block and directory index pool are flushed, thus invalidating the buffers associated with the current UCB. This invalidation is performed only on a single node because the sequence numbers associated with the serialization locks protect these buffers in a VAXcluster.
- The special file write virtual function also scans through all the BFRDs in the cache, discarding those in the specified pool for the current UCB when the user writes to the following special files:
  - Index file
  - Storage bitmap file
  - Quota file
  - Directory

  For more information about how writing to these special files is done in a VAXcluster system, see Chapter 8.
- When the quota file is deaccessed, all the quota file blocks that were modified are written out to disk after the quota cache is cleared. Then the buffers associated with the quota file data blocks are purged.

4.6.2 Changing the LBN of a Buffer

When the index file header is modified (in the CREATE_HEADER routine), the LBN of the buffer is changed by the RESET_LBN routine to reflect the alternate index file header. This modification ensures that a file header will be available when a block is written to disk. The buffers containing the alternate index file header are immediately invalidated to avoid confusion.

A similar operation is performed when the index file header is either read or extended (in the READ_IDX_HEADER routine). If the file size from the header is incorrect, the alternate index file header is read, and the LBN of the buffer is changed. If either operation fails, the buffer is simply invalidated. Likewise, the LBN of a buffer must be changed when the index file itself is extended.

The RESET_LBN routine is also used when a contiguous file is copied to extend it. It performs the following actions:

- Reads the blocks of the old file as data blocks
- Changes the LBNs associated with the buffers
- Writes the blocks explicitly

The blocks may remain in the cache because the LBN recorded matches their new location. This operation is also performed when a directory is compressed.

Chapter 5
The ACP Functions

Fudd's Law of Insertion *What goes in, must come back out.*
Van Mizzel, Jr.

Agnes Allen's Law *Almost anything is easier to get into than out of.*
Agnes Allen

Outline

5.1 Introduction

Accessing data on disk is controlled by the ACP functions and function modifiers, which are part of the QIO interface to the file system. The seven main functions are as follows:

- Accessing a file
- Creating a file
- Deaccessing a file
- Modifying a file
- Deleting a file
- Mounting a disk[1]
- ACP control (disk quota adjustments, dismounting a disk)

These virtual I/O functions may be invoked by the $QIO system service:

- Directly by the user
- Implicitly as part of an RMS operation
- As a result of the MOUNT command
- As a result of using the System Management Utility (SYSMAN)[2]

5.2 ACP-QIO Interface

The file system data structures are represented by on-disk information. The XQP replaces the metadata (that is, the information about the file) on disk with in-memory copies or representations of the same things—WCBs, FCBs, VCBs, and so on.

All the functions require access to the file header. For example, the functions must read the header or alter it. An open operation (which is expressed as an access function with access modifier; without the access modifier, it is a lookup function) takes the map data from the header and represents some of it in a window control block in the privileged I/O database.

---

[1] The XQP does not perform the entire mount. The SYS$MOUNT service does most of the work and sets up file system data structures; the IO$_MOUNT QIO executed by the XQP simply verifies and acknowledges the mount.

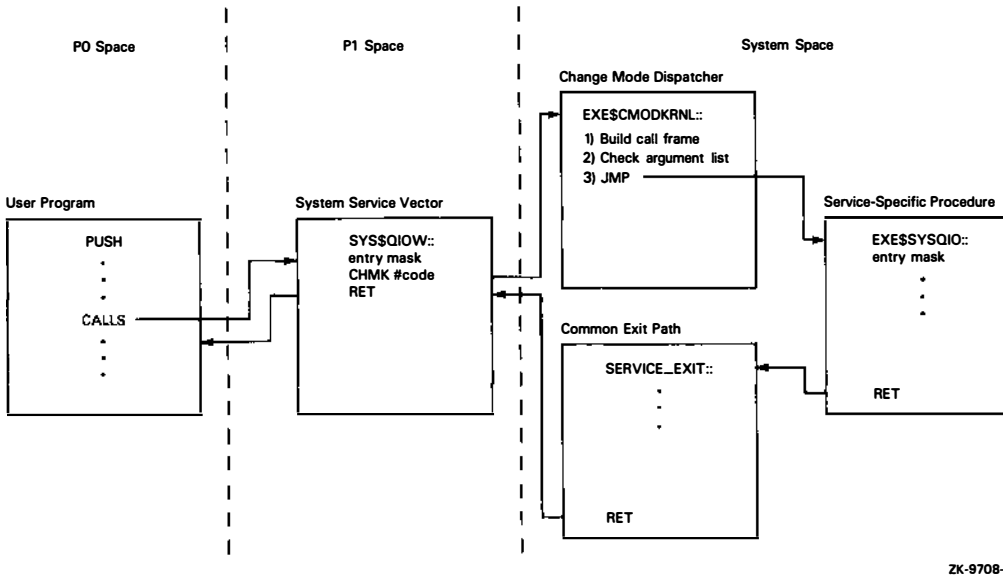
[2] SYSMAN includes the DISKQUOTA command set, which operated as a standalone utility in VMS Version 4.6.

All the major functions are executed by the XQP, where they are translated to logical read and write QIO functions. In the normal case, read and write requests go through the SYSQIOREQ module, not through the XQP, which must reference WCBs, VCBs, and FCBs. If the necessary disk information is mapped and if the access is legitimate, the $QIO request may bypass the XQP altogether. In fact, this is the desired outcome.

In a sense, the XQP merely takes care of the housekeeping functions (the metadata-establishing functions), and the XQP code is often avoided. For example, the major ACP functions are usually only performed once: once per file open, per file remap, and so on. And because I/O is a limiting factor, data is often represented with absolute data structures such as cathedral windows (which avoid the remap function).

Figure 5–1 shows how data travels from the user through the I/O subsystem to the XQP.

Figure 5–1: ACP-QIO Interface



When performing I/O, the file system is awakened in a manner that is appropriate to its environment. With an ACP, a $WAKE is issued to the ACP. On the other hand, the XQP is entered when a kernel-mode AST is queued, and control is then transferred to the XQP dispatcher.

In either case, the following four steps are performed:

1. Getting the request
2. Dispatching to the main operation
3. Posting the results of the operation
4. Returning resources

5.2.1 Getting the Request

When the user issues a $QIO system service request with a main function and a function modifier, the $QIO service and its associated **function decision table** (FDT) routines verify the correctness of the parameters passed and construct a **complex buffer packet**, or ABD. The routine EXE$QXQPPKT is then called, which queues a kernel-mode AST to the process with the following two pieces of information:

- The XQP entry point F11B$L_DISPATCH as the AST routine address
- The IRP address as the AST parameter in the call to SCH$QAST

The routine DISPATCH queues the IRP to the tail of the XQP work queue and returns if a request is currently in progress. This can be determined by checking whether IO_PACKET (in the XQP impure area) has a nonzero value. If there are no packets in the queue, DISPATCH performs the following actions:

- Prohibits process deletion
- Changes the access mode of the I/O channel to kernel
- Switches to the XQP private stack as the kernel stack
- Saves the frame pointer
- Calls the main XQP routine DISPATCHER

At this point, DISPATCHER takes the following three actions:

- Finds the (next) request on the work queue

  The DISPATCHER routine first calls the routine GET_REQUEST to remove an IRP from the XQP work queue. An ACP maintains its work queue using a queue header in the system pool. The queue header of the XQP work queue is part of the **XQP impure area** in P1 space and is therefore process specific.

- Validates the information

  After DISPATCHER dequeues the request packet, it performs the necessary validation. It initializes the value of the per-operation performance counters and the common data area (the impure area). DISPATCHER also checks the device control blocks that the request references to ensure they are valid and that the device is mounted. In addition, it validates the user-supplied parameters to make sure they are consistent with each other for an operation. For example, an extend operation and a truncate operation at the same time are not allowed.

  DISPATCHER sets up UCB and VCB pointers to point to the correct device, and then assigns the I/O channel to the UCB. If the operation involves a volume set, DISPATCHER also assigns a pointer to the RVT of the current volume, and obtains the relative volume number.

  A local copy of the user's privileges is made and is used for all subsequent privilege checks. The SYSPRV bit is set in this local copy if any of the following conditions is true:

  - The user has a UIC group less than or equal to MAXSYSGROUP.
  - The user is the owner of the volume.
  - The user's UIC group matches the volume owner group and the user has GRPPRV privilege.

  Also, the following cleanup flags are set:

  - CLF_VOLOWNER or CLF_GRPOWNER, if the user is the owner of the volume or has GRPPRV and the volume group matches the owner's group
  - CLF_SYSPRV, if the user has BYPASS, READALL, or SYSPRV privileges

- Sets up the impure area

  After GET_REQUEST zeroes the impure area to clear the fields from the previous operation, it sets up pointers to the control blocks that are relevant for the operation. For example, if the operation is on an open file, pointers are set up to point to the file control block and to the window control block so that access to all of the relevant data structures is established.

5.2.2 Dispatching the Operation

There is a major function routine for each of the major functions, and every function involves the following steps:

1. Setting up the operation.

   An important part of setting up the operation is obtaining the required number of buffers, or **buffer credits**, to perform the operation. The routine GET_REQD_BFR_CREDITS ensures that the process has sufficient buffer credits to proceed with any function. The buffer credit quota mechanism prevents a process from stalling for lack of available buffers while it holds a lock on a critical system resource.

   This routine checks buffer credits for all three buffer pools, and if they are sufficient for the operation, it releases the cache lock. If there is an entry waiting in the ACP queue block, an AST is queued to resume the stalled thread for that process. If there are not enough credits, the process is inserted in the pool wait queue and stalled with the usual WAIT_FOR_AST–CONTINUE_THREAD mechanism.

2. Performing the requested operation.
3. Cleaning up after the operation.
4. Lowering the IPL to allow process deletion.
5. Restoring the stack.

This approach is followed because many cleanup functions should be performed in a centralized manner. When these functions are done at the end of processing, they must be done only once. However, potentially, they might need to be done many times during the course of processing a request. For example, functions such as performing checksum operations on file headers and reinitializing the file control block are usually done more than once while a request is being processed.

5.2.3 Posting the Results

The following actions occur after the request has been processed:

- The complex buffer packet is modified; any buffers that need to be returned to the user are written back to the complex buffer packet.
- The completion status is placed in the I/O packet.
- The I/O packet is sent to the VMS common I/O post facility. Here, the complex buffer is disassembled, and the individual pieces are revalidated and written back to the user buffers from which they came.

Although necessary in the ACP environment, copying buffers back to the user is not strictly necessary in the XQP environment because the XQP executes in the context of the requesting process. The same mechanism is used for both the ACP and the XQP to simplify the QIO interface. Also, because the XQP does not access user buffers directly, address probing is not necessary.

5.2.4 Returning Resources

Once the file operation has been completed, any resources are returned. The file system uses two major resources when it is processing a request:

- Locks—Any locks that were taken out to synchronize the operation or to protect data are released.
- Buffers—Any buffers that were allocated for this operation are released.

5.3 Major ACP Functions

The file system ACP functions are well-structured. Utility routines exist within the file system that the file system itself uses, so a map of what routines call other routines is meaningless.

ACP function execution can be divided into four levels:

Level	Description
Dispatch level	Handles user input and dispatches requests. This level consists of modules such as SYSACPFDT (in the SYS facility) and DISPATCH and DISPAT (in the F11X facility).
Function level	Consists of the top-level ACP function routines such as ACCESS and CREATE.
Synchronization level	Controls access to file system entities by means of locking, FCB invalidation, system blocking routines, and other routines in the LOCKERS module.
Utility routine level	Performs the low-level simple functions such as reading a file header (READ_HEADER), writing a file header (WRITE_HEADER), or establishing a connection to the quota file (CONN_QFILE).

There is a separate top-level routine in the file system for each of the major functions:

Function	Meaning
Access	Opens a file or looks up a file (establishes that it exists). It also retrieves information from a file. For example, a directory lookup to take a file name and return the file ID would use the access function. The IO$M_ACCESS modifier establishes an access path to the file. For more information, refer to Section 5.3.1.
Create	Creates a file. It also creates a directory entry, either for the file that is being created or for an existing file. The IO$M_ACCESS modifier establishes an access path to the file. For more information, refer to Section 5.3.2.
Delete	Deletes a file. It can also remove the directory entry for the file that is being deleted. For more information, refer to Section 5.3.3.
Modify	Changes the characteristics of an existing file, which may involve extending it, writing attributes to it, or truncating it. For more information, refer to Section 5.3.4.
Deaccess	Deaccesses a file. For more information, refer to Section 5.3.5.
ACP control	Consists of miscellaneous functions. For more information, refer to Section 5.4.

5.3.1 Access Function

Part of the QIO interface to the XQP is the ACP function IO$_ACCESS. This function searches a directory for a specified file and accesses the file, if it is found. It can take two function modifiers:

- IO$M_CREATE—Creates the file if it does not exist.
- IO$M_ACCESS—Opens the file on the user's channel.

Because the function is IO$_ACCESS, the routine ACCESS is called. The access function performs the following tasks:

- Finds the directory entry, if necessary.
- Serializes processing to the file, allowing stable FCBs to be found or created. Establishing serial access to the file and to the FCB chain is important because it prevents another process from changing the information in the FCB chain while the chain is being scanned. Serializing access to the file also prevents the file itself from changing state (including protection, size, access by other users) while the current request is in progress.

- Locates the file control block. If the wrong lock basis was serialized on (that is, if the user was trying to access an extension header directly), this problem is detected. In this case, the serial lock is released, and the correct lock basis (for the primary header) is serialized on.
- The file header is read, and a file control block is created if one was not found.
- The primary FCB is created if one is not found.
- Checks access conflicts, and obtains the access lock in the appropriate mode. For more information on the access lock, refer to 8.3.2.
- Creates a window to the file. The routine MAKE_ACCESS threads the window onto the FCB and updates the access counts.
- Sets the WCB$V_WRITE_TURN bit for directories and other special files (such as QUOTA.SYS) whose contents may be contained in the XQP cache. Write operations to such files cause the cache contents to be invalidated.
- Checks the expiration date.
- Builds and validates extension FCBs
- Checks and audits user access to the file, if necessary.
- Flushes the caches if the file is a special one.
- Obtains the appropriate cache lock.
- Reads attributes if they were requested.
- Determines the need for cathedral windows.

5.3.2 Create Function

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

The function modifiers are:

- IO$M_CREATE—Creates a file.
- IO$M_ACCESS—Opens the file on the specified channel.
- IO$M_DELETE—Marks the file for deletion when it is created. No directory entry for the file is made because it is only a temporary file.

If the modifier IO$M_CREATE is specified, a file is created. The file ID of the file created is returned in the **file information block** (FIB). If the modifier IO$M_DELETE is specified, the file is marked for deletion immediately.

If a nonzero directory ID is specified in the FIB, a directory entry is created. The file name specified by the user is entered in the directory, together with the file ID.

The create function is invoked if an IO$_CREATE function code is specified, or if DISPATCHER detects SS$_NOSUCH FILE from ACCESS when IO$M_CREATE (create_if) was specified. The create function performs the following basic steps:

- Performs a cleanup operation from a failed access attempt if this is a create-if operation.
- Performs a write access check on the parent directory because the FIND routine within the access operation only checked execute access.
- Finds the volume for the file.
- Checks the user's rights to create files on that volume.
- Creates a file header.
- Creates the primary FCB.
- Enters the file (if it is not a temporary one) into the specified directory.
- Serializes access to the previous version of the file if attributes are to be propagated. The FCB list is searched, and an FCB is created, if necessary. Also, attributes are copied to the file.
- Determines the back link.
- Writes the updated header to disk.
- Performs write-attribute processing, and updates any ACLs to include the creator.
- Charges quota for the file (if quotas are enabled).
- Accesses the file if requested.
- Extends the file to the desired length.
- Updates the file header chain with the ACL.
- Remaps the file if it was extended (and if cathedral windows were specified).
- Deletes any file that was superseded or removed (which is a cleanup function).

5.3.3 Delete Function

Delete file is a virtual I/O function that removes a directory entry or file header from a disk. It can take one function modifier, IO$M_DELETE, which deletes the file (or marks it for deletion).

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

If IO$M_DELETE is not specified, the directory entry only is removed.

If a nonzero directory ID is specified in the FIB, a lookup subfunction is performed. The file name located is removed from the directory.

The DELETE function is invoked if an IO$_DELETE function code is specified. It performs the following basic steps:

- Finds and removes the directory entry.
- Serializes access to the file, and reads its header.
- Seeks for and creates the FCB if necessary.
- Checks if the directory entry removed was the primary entry. If it is not, the file itself is not deleted. A directory entry is considered the primary entry if the directory file ID matches the file's back link and the name in the directory matches the name stored in the header.
- Checks for delete access to the file.
- Checks to guarantee that the file is empty if the file is a directory.
- Audits the deletion if required.
- Checks for other accessors.
- Marks the header for deletion.
- Returns any cached buffers for the file (directories only).
- Marks the FCB for deletion.
- Deletes the file if this process is the only accessor.
- Releases the access lock (manipulated when checking for other accessors).
- Deletes any FCBs.

5.3.4 Modify Function

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

The function code IO$_MODIFY takes no function modifiers. It is used for modifying the characteristics of an existing file, which means extending it, writing attributes, or truncating it.

The MODIFY function is invoked if an IO$MODIFY function code is specified. It performs the following basic steps:

- Locates the directory entry if required
- Serializes access to the file
- Seeks for and creates the FCB if necessary
- Interlocks against other accessors
- Checks access to the file
- Performs and audits write-attributes processing if necessary
- Performs extension or truncation
- Updates the file header chain

5.3.5 Deaccess Function

Deaccess file is a virtual I/O function that deaccesses the file. It has no function modifiers.

Because the function is IO$_DEACCESS, the routine DEACCESS is called. It performs the following basic steps:

- Serializes access to the file.
- Rebuilds the FCBs if any operation must be done to the file.
- Requests cleanup deletion of the file if it is marked for deletion and this is the last access.
- Updates the revision count.
- Updates the file highwater mark.
- Clears the deaccess lock flag if attributes are being written.

- Writes the attributes.
- Performs any requested truncation. If this process is not the only accessor, delayed truncation is performed. However, if this process is the last accessor and delayed truncation was requested, truncation is performed at this time.

The ERR_CLEANUP routine actually deaccesses the file; it is is invoked after the DEACCESS routine exits.

5.3.6 ACP Control Functions

Miscellaneous control functions are performed by the ACP control functions. All ACP control functions originate from the ACPCONTROL routine.

The following table lists these control functions and what they do:

Function	Description
Remap	The REMAP function invokes REMAP_FILE under the file serialization lock to completely map the file.
Lock volume	The LOCK_VOL function locks the current volume by taking the blocking lock for the volume (TAKE_BLOCK_LOCK).
Unlock volume	The UNLK_VOL function clears the VCB$V_NOALLOC bit, establishes the free space value clusterwide (referenced in the free space allocation lock block field), and dequeues the blocking lock.
Mount verification	The FORCE_MV function forces a shadow set virtual unit through mount verification to ensure that the SCB is consistent and that shadowing information is updated throughout the cluster. This function can be requested by a process with SYSPRV.
Mount	The MOUNT function informs the file system when a volume is mounted. This function is part of the volume mount procedure only, so users cannot access it directly. For more information on the volume mount procedure, see Section 3.3.

Function	Description
Disk quota	Enables disk quota enforcement on a volume or volume set. This function includes two types of operations: • To enable and disable the quota file itself • To manipulate individual quota entries For more information on disk quota operations, see Section 5.4.1.
Dismount	Flushes all caches and marks the SCB as the volume being dismounted. The interesting aspect of this operation is that the SCB is possibly written to (asynchronously) by mount verification. The SCB I/O must be retried to make sure that a consistent SCB is read or written.

5.4 Miscellaneous File System Requests

The major ACP functions are represented by QIO functions. Moreover, multiplexed under those are a variety of operations that are distributed across those major functions. A number of these other functions may occur as a result of any of several major functions. Although some of these file system functions are performed explicitly, others are performed implicitly on behalf of the user.

Miscellaneous file system functions include the following:

- Disk quota operations
- Directory manipulation
- Space management
- Attribute handling
- Bad block processing

5.4.1 Disk Quota Operations

Some quota operations are implicit, in that when space is allocated or deallocated, the user's quota is automatically adjusted.

Disk quota enforcement is enabled by a quota file on the volume, or on relative volume 1 if the file is on a volume set. The quota file appears in the volume's master file directory under the name QUOTA.SYS;1.

Operations that may be performed on the quota file include the following:

- Enable
- Disable
- Add entry
- Examine entry
- Modify entry
- Remove entry

5.4.1.1 Quota File Operations

A user can request that various ACP control functions be performed on the quota file. These quota operations involve explicit manipulation of the quota file, and they are called by the System Management Utility (SYSMAN)[1]. The quota file is never manipulated directly by VMS because its individual entries are cached by the file system. Rather, the operations interlock with the cache, so consistent results are always seen.

These user-invoked quota operations are performed by the QUOTA_FILE_OP routine, which handles enabling and disabling quota processing, and adding, examining, modifying, and deleting quota file entries.

The XQP synchronization rules dictate that the file serialization lock must be acquired before the volume allocation lock. The lock basis for the quota file is contained in the quota file FCB, which is located by means of the pointer VCB$L_QUOTAFCB. However, the allocation lock protects the quota file FCB; that is, VCB$L_QUOTAFCB is not stable except under the allocation lock.

The solution is for QUOTA_FILE_OP to perform the following loop:

1. Request the serialization lock on the quota file (using whatever random value it gets by looking at VCB$L_QUOTAFCB)
2. Take out the allocation lock
3. Check that VCB$L_QUOTAFCB matches
4. Unlock and retry until it does

However, an additional aspect of these quota operations is that the blocking lock is requested for an add quota function. DISPATCHER does not request the blocking lock for ACP control functions because some affect the blocking lock state.

QUOTA_FILE_OP also checks protection and permission to the file (by privilege or access).

---

[1] SYSMAN includes the DISKQUOTA command set, which operated as a standalone utility in VMS Version 4.6.

The following table describes the quota operations and their functions:

Quota Function	Description
Enable	Connects to the quota file. In other words, it enables quota for the volume.
Disable	Flushes the quota cache, forces writes of any quota file buffer blocks, and then performs the actual quota file deaccess (DEACC_QFILE).
Examine	Returns a quota file record.
Add entry	Finds the next free record, writes the quota information, and calls the EXTEND_CONTIG routine if it is necessary to enlarge the file.
Modify	Changes an entry and writes it. The process must hold the volume blocking lock to modify the usage figure.
Remove entry	Returns an old entry. The entry is zeroed under an exclusive lock on the quota entry.

5.4.1.2 Quota Cache

The quota cache has entries based on UICs to keep track of allowed usage, current usage, and so on, without having to read and write the QUOTA.SYS file itself all the time. The quota cache is allocated in nonpaged pool by MAKE_DISK_MOUNT in the MOUNT module MOUDK2 and deallocated by CHECK_DISMOUNT.

The quota cache is found by following the VCA block pointer in the VCB$L_QUOCACHE field.

Each quota cache entry contains the following information:

- UIC
- Quota information (usage, permanent quota, overdraft)
- Lock status
- Block used with the quota cache entry locks
- Quota file record number
- LRU indexes

The cache header contains an LRU counter. When a new entry is added, the value is put into the entry, and this counter is incremented.

The following routines operate on the quota cache:

Routine	Function
FLUSH_QUO_CACHE	Flushes all entries to disk. The corresponding record on disk is located and updated (by CLEAN_QUO_CACHE). Any quota entry locks are released, including a conversion to null mode of the quota cache lock itself.
SCAN_QUO_CACHE	Finds an entry in the cache. If the cache is marked invalid, this routine tries to get the normal protected read cache lock to enable the cache. If the entry is not valid, the quota entry lock (protected write) is obtained to make it valid and to get the current quota values from the lock value block. If the entry is not found in the cache, it is added to the cache (either by LRU replacement or by expansion). SCAN_QUO_CACHE also sets the VCA$V_CACHEFLUSH flag in the cache header if it finds the quota cache invalid and cannot obtain the cache lock. The CLEANUP routine checks for this condition and flushes the quota cache when processing is finished (to reflect the changes to the process holding the quota cache lock).
CLEAN_QUO_CACHE	Updates the disk record from a cache entry. The disk buffer is marked as dirty, and the cache entry is marked as clean.
ENTER_QUO_CACHE	Copies a given record into the cache. The LRU index is updated if requested, and the entry is marked as dirty if requested.

5.4.1.3 Accessing the Quota File

Enabling quota processing on a volume makes a call to CONN_QFILE. The CONN_QFILE routine establishes a connection to the quota file and calls the FIND routine to locate the quota file. CONN_QFILE performs these additional actions:

- Finds or creates the FCB under the quota file serialization lock
- Builds the extension FCBs
- Requests write access for the quota file
- Allocates the quota cache, linking it to the VCB
- Sets up the ACBs in the cache header for the various blocking routines
- Takes out the quota cache lock if the quota file is already write-accessed

5.4.1.4 Processing the Quota File

The quota file is a sequential contiguous file with no version limit, and quota file records are fixed-length 32-byte records. The main routine for quota file processing is SEARCH_QUOTA. This routine locates a quota record for a given UIC.

SEARCH_QUOTA scans the quota cache, updating the quota file from the cache entry if necessary. It scans the quota file if the record cannot be found, or if a wildcard search was requested.

If a wildcard search was specified, the scan of the quota file is performed before the scan of the quota cache to return the records in order by UIC. If the record returned is in the cache, the returned address is that of DUMMY_REC within CHARGEQ. This value takes on special-case status in other places within CHARGEQ.

REAL_Q_REC is the address of the buffer containing the actual disk quota record, if there is one. WRITE_QUOTA updates the cache entry and the disk record (in which case the buffer is marked as dirty) depending on these variables.

CHARGE_QUOTA performs the system processing of charging for quota, checking for overdrawn quota, and so on. It writes out the new quota record if the quota charge is valid.

5.4.1.5 Deaccessing the Quota File

The inverse of connecting the quota file is done by DEACC_QFILE, which deaccesses the quota file. It starts by returning any quota file buffers (in KILL_BUFFERS). It performs these other actions:

- Demotes the access lock on the quota file to show that the process has deaccessed the file
- Deallocates the quota cache
- Releases the quota cache lock if it was taken out

5.4.2 Directory Manipulation

Directory manipulation is an implicit file system function in many ways. Several of the file functions result in a directory search to look up a file. For example, in the case of a delete function, a directory entry is removed. In the case of a create function, a directory entry may be created. This type of directory processing is all handled with a common mechanism.

Directory manipulation is done with a common set of directory routines. First of all, the directory file must be located, so there is a process for implicitly accessing the directory. A window does not have to be built because directories are always contiguous. As a result, there is enough information in the file control block to find the directory on disk.

Because there is bounded access control information, the file system keeps file control blocks for recently referenced directories, which is a performance optimization. Repeated references to the same directory eliminate the I/O bottleneck of having to go to the disk to read the file header for the directory file every time a directory operation is requested for that directory.

There is a common directory scanner routine that does all the directory processing for all the components of the file system. This scanner implements wildcard searching so it can find the next occurrence of the specified wildcard string. It accepts input pointers to specify a point at which to start scanning the directory, and it returns the output pointers of where it has stopped scanning. It also returns pointers to the previous directory entry. The interface to this scanner is enormous because it is used by so many different facilities in the file system.

Once a directory has been scanned, several conditions can result:

- For a directory access, the scanner obtains the directory entry and returns the file ID.
- For a create function, the file system either has the pointers to where a file should be created or the pointers to a previously existing directory entry that is to be superseded.
- For a delete function, the scanner returns the pointer to the directory entry that is to be removed.

These operations are then performed by separate routines.

5.4.3 Space Management

Space management is another common mechanism. Any of the following functions can result in either allocating or deallocating disk space:

- The extend and truncate routines operate on the file header and call the common allocation and deallocation routines.
- The allocation routines manipulate the storage bitmap as well as some of the allocation caches. They are capable of several different types of allocation.

  The default is random allocation, which is based on the assumption that most file activity is dynamic. That is, for files that are written once and may be read once, or never, or twice, it is more efficient to do the allocation as quickly as possible without trying to optimize file placement. As a result, these routines simply take whatever disk blocks are readily available at hand and allocate them to the file being created or extended.

  For files that have a greater degree of permanance, there are several other options:

  - Contiguous allocation guarantees that all of the space allocated is in one area.

- Contiguous-best-try allocation specifies that the allocation is contiguous if possible (within three tries).
- Placed allocation specifies that the first attempts at allocation occur at a specified location on the volume. That is very useful for co-locating different portions of an active database.

5.4.4 Attribute Handling

Attribute handling interprets the attributes that are specified in the complex buffer packet, and it involves either the reading or writing of attributes. For instance, the access function causes attributes to be read from the file back to the user. The create, modify, and deaccess functions result in attributes being written or propagated to the file.

A number of file attributes are protected. That is, they can be read but they can only be written in a controlled way.

However, some attributes cannot be written at all. This restriction is primarily to protect the integrity of the file system. These attributes are represented as follows in the file header field FH2$L_FILECHAR:

Bit Name	Meaning
FCH$V_CONTIG	The file is contiguous. This bit can only be cleared by the user. It is possible for a user to take a file that has been marked contiguous and mark it noncontiguous. However, the file system does not allow the reverse to be done (that is, it is not possible for a user to mark a file as contiguous).
FCH$V_SPOOL	The file is spooled. The spool bit is a special case for the internal handling of spooled files on transparently spooled devices. The file system does not allow a user to modify this bit.
FCH$V_BADBLOCK	The file contains a bad block. It indicates that bad block processing is to be performed on the file at a later time, such as after it is deleted. For more information, see Section 5.4.8.
FCH$V_NOCHARGE	The file space is not charged against quota. The nocharge bit suppresses charging the space of a file to its owner's quota. One of the last functions performed in an XQP request is to reflect new disk usage in the quota file. This is done unless FIB$V_NOCHARGE is set (a user cannot set this bit because GET_FIB clears it).

Bit Name	Meaning
FCH$V_MARKDEL	The file is marked for deletion. The mark-for-delete bit is used for the internal purposes of the file system, so a user cannot modify it directly.

5.4.5 Dynamic Highwater Marking

Disk scavenging is a security problem where a user allocates disk space and then searches it for interesting contents of previous files that have been deleted. VMS solves this problem with the combination of the two following techniques:

- Erase-on-allocate
- Highwater marking

Both are enabled when the highwater marking volume attribute is enabled with the SET VOLUME/HIGHWATER command.

VMS maintains a **highwater mark** which indicates how far the file has been written in its allotted space on the disk. All blocks in the file up to the highwater mark are guaranteed to have been written since they were allocated to the file. The user is not permitted to read beyond the highwater mark, and thus cannot read stale data from the file.

Erase-on-allocate is the more costly but conservative technique. It is used when the file is open, allowing any form of shared access or nonsequential access. Erase-on-allocate, as its name implies, simply means erasing all disk blocks when they are allocated to the file. The file's highwater mark is set to point to the end of the newly allocated and erased space.

Highwater marking is used only when the file is open for write with exclusive access in sequential-only mode.[1] In this mode, the highwater mark is maintained in memory and cannot be maintained across multiple nodes of a cluster with acceptable performance (which is why access is limited to a single accessor).

This is a restrictive but common set of circumstances that allow the file system to perform the following functions:

- Maintain a valid highwater mark dynamically
- Incur erase operations only under unusual circumstances

Sequential-only access is specified to the XQP with the FIB$V_SEQONLY bit in an IO$_ACCESS function. It is a declaration that the file is a sequential file, but it does not enforce any ordering of disk I/O. This type of access contributes to the complexity of the highwater marking logic.

[1] This mode of access results when a sequential file is opened for write through RMS with the usual defaults.

Sequential-only access is an enabling factor in highwater marking to prevent possible corruption of nonsequential file organizations (such as indexed files) during a system failure. The highwater mark is maintained in the FCB while a file is open and is written to disk only when the file is closed. Thus, should the system fail while the file is open, the blocks written past the last-recorded highwater mark will be lost.

This loss can be tolerated and controlled with sequential files, but it can cause arbitrary corruption in nonsequential file organizations. Nonsequential files are therefore opened without FIB$V_SEQONLY and are handled with the **erase-on-extend** technique.

5.4.5.1 Basic Highwater Mark Algorithm

The basic principle of highwater marking is not to allow the process to read what has not yet been written. The highwater mark indicates the highest block that has been written in the file. If the user attempts to read past the highwater mark, the read is stopped. No error status is returned; as far as the user is concerned, the read has completed successfully.

When a write function occurs that extends past the current highwater mark, the highwater mark is updated to reflect the point at which the write operation ended.

Finally, when a write occurs that begins beyond the current highwater mark, the blocks between the current highwater mark and the start of the write function are erased to fill the gap that would otherwise be left in the file. The code that implements this simple algorithm is deceptively complex because the VMS I/O system allows multiple concurrent read and write functions whose interactions must be controlled.

5.4.5.2 Highwater Mark Handling Routines

Most highwater mark processing is handled by the executive I/O routines in SYSACPFDT and IOCIOPOST. Certain exception cases are handled by the module RWVB in the XQP, along with the other exception cases of virtual I/O.

Highwater mark processing begins in the IOC$MAPVBLK routine. This routine enforces the highwater mark for read operations by not mapping blocks past the highwater mark. Thus, if a read QIO extends past the highwater mark, it is treated as a segmented read, and the following actions occur:

- The portion of the file that lies within the highwater mark is mapped and is executed by the driver.
- The IOCIOPOST routine then attempts to map the remaining portion, as it would for any segmented I/O.
- Total map failure occurs, which causes the IRP to be sent to the XQP.

In this circumstance, IOC$MAPVBLK indicates that the IRP starts past the highwater mark by setting the flag IRP$V_START_PAST_HWM. This flag causes the RWVB routine to recognize the IRP as a read-past-highwater IRP, and it immediately returns the IRP to the user.

Write operations past the highwater mark are permitted, and they are mapped by the IOC$MAPVBLK routine. However, a write-past-highwater operation requires the highwater mark to be updated. This checking is done by IOC$CHECK_HWM, located in SYSACPFDT.

While a write-past-highwater operation is in progress, the highwater mark is in transition; until the write completes, it is impossible to know whether the blocks affected by the write operation have been written or not. Therefore, the file system does not permit read operations into the transition region of the file.

Concurrent reads and writes are managed by having two highwater marks for the file:

- FCB$L_HIGHWATER—Is the current highwater mark. All blocks up to this location are known to have been written and therefore may be read.
- FCB$L_NEWHIGHWATER—Is the new highwater mark. All blocks between this location and FCB$L_HIGHWATER are in the process of being written.

Each write operation that goes past the current value of FCB$L_NEWHIGHWATER causes its value to be updated. The following two fields are also modified:

- The IRP is flagged with the bit IRP$V_END_PAST_HWM to indicate that it affects the highwater mark.
- The counter FCB$W_HWM_UPDATE is incremented to reflect the outstanding update.

While writes are in progress in the transition region, reads cannot be allowed in the transition region because the file system does not know which blocks have or have not been written. Such reads must be stalled until the writes complete and the highwater mark is again stable. This stalling is done by queuing read-past-highwater IRPs onto the end of the FCB$L_HWM_WAITFL queue.

Completed writes flagged with the IRP$V_END_PAST_HWM flag are processed by the routine HWM_END in IOCIOPOST. This routine performs the following actions:

- Decrements the pending write counter. If the counter goes to zero, it means that all blocks in the transition region have been written and that the new highwater mark is stable.
- Copies FCB$L_NEWHIGHWATER into FCB$L_HIGHWATER.
- Dequeues any pending reads on the wait queue, and reprocesses them by sending them through the IOC$QNXTSEG routine.

Write operations that start beyond the new highwater mark cause considerably more trouble. If such a write were simply executed, it would leave a gap in the blocks written to the file. This gap must be filled by erasing the blocks between where the FCB$L_NEWHIGHWATER field indicates and where the write actually begins. This type of write operation is flagged with the bit IRP$V_START_PAST_HWM, and the file system performs the following actions:

- Copies the starting VBN of the gap (represented by the current value of FCB$L_NEWHIGHWATER) into IRP$L_ERASE_VBN.
- Increments the pending erase counter FCB$W_HWM_ERASE.
- Queues the IRP to the XQP routine RWVB, which uses the routine ERASE_VIRTUAL to erase the appropriate file blocks.

While the erase operation is in progress, asynchronous writes to the transition region of the file cannot be allowed because there is no way to guarantee whether the write operations or the erase operation will be processed first by the disk driver. As a result, while the pending erase count is nonzero, write operations into the transition region are likewise stalled on the FCB$L_HWM_WAIT queue. Pending read operations are stored on the back of the queue (that is, in the FCB$L_HWM_WAITBL field), and pending writes are stored on the front of the queue.

After the RWVB routine has executed the erase operation, the following actions occur:

- Its subroutine END_HWM_ERASE decrements the pending erase counter.
- If the counter goes to zero, RWVB pulls any pending write operations (but not reads) from the front of the stall queue and restarts them by means of the IOC$QNXTSEG routine in a manner similar to IOC$HWM_END.
- Finally, the write that incurred the erase operation is sent to the driver through the REQUEUE_REQ routine. This write operation is now a normal write-past-highwater operation and is treated as one.

Additional complexity can be introduced by undisciplined software that attempts to write past the end of the file's allocated space. Because the file can also be extended asynchronously, such a write only extends the highwater mark to the end of the allocated space (where the write operation will stop if the file is not extended), and its IRP is flagged with IRP$V_PART_HWM.

If the file is extended while the I/O is in progress, a subsequent trip through IOC$QNXTSEG will map the new blocks. The IRP$V_PART_HWM flag indicates that the IRP must be revalidated with another call to IOC$CHECK_HWM before proceeding.

Finally, write-past-highwater operations that encounter an I/O error require special handling. If the write contained multiple blocks, some blocks usually remain unwritten because the I/O was stopped short by the error. Because the advance of the highwater mark has already been committed, these blocks must be overwritten to avoid having unwritten blocks in the file. This is accomplished (in RWVB) by erasing the unwritten blocks of the transfer before returning the error to the user.

5.4.6 Spool File Processing

A **spool file** is a file without a directory entry that is flagged as spooled. It is sent to the symbiont when it is deaccessed. The idea behind spooling is to allow a process to pretend it is writing to a printer, when in fact it is writing to an intermediate file.

The IRP$L_UCB field (which is loaded into the impure cell CURRENT_UCB by the GET_REQUEST routine) refers to the spool file. IRP$L_MEDIA is set to the spooled device UCB (a printer). Spool file operations are recognized when a process sends a create function to a printer specified as spooled, and this function is then translated into the creation of a spooled file.

Requests to operate on spool files are recognized in FDT processing. The file name user buffer is replaced by the user name and account to become the file name in the header. GET_REQUEST then notices that IRP$L_UCB is different from IRP$L_MEDIA, and sets the cleanup flag CLF_SPOOLFILE.

The spool flag is set in the file header for spool files by the CREATE routine. This flag causes the FILL_FCB routine to set the FCB$V_SPOOL bit in the file control block. This flag is one of the characteristics that cannot be changed by WRITE_ATTRIB.

When a spool file is deaccessed, the IO$_DEACCESS FDT processing transmits the caller's user name in the P2 string buffer of the complex buffer packet. The user name is used by an ACP-type file processor to submit the queued request under the right user name; it is not used by the XQP because the XQP is running in the caller's process.

In the file system, the CLF_DOSPOOL cleanup flag is set, which causes a queue request to be sent to the job controller containing the following information:

- File ID
- Filename string constructed from the device name and the name in the file header
- Queue name taken from the VCB of the spooled device

If the symbiont request fails, the file is deleted (by setting CLF_DELFILE), and the job controller error status is returned to the user (in USER_STATUS).

5.4.7 Access Control List Processing

Access control lists (ACLs) are also managed through attribute handling. There are several attribute control codes that are used to read either the entire access control list or selected entries, to update the access control list, and so on.

The access control list is contained in the ACL area of a file header. Although an ACL can be relatively large, unrestricted use of access control lists forces multiheader files to be created—an important performance consideration. A file header is 512 bytes long, and when either the access control list or the map pointer space exceeds this length, extension headers must be created and chained to the original header to control the overflow.

The address of the ACL for a file is a parameter passed to the CHECK_PROTECT routine. It is created (copied from the file header chain) only during initial FCB creation.

For an open file, the access control list is maintained in paged pool, and it can be located from the ACL queue of the ORB, which is located in the primary FCB for the file. When an ACL is processed, the file system actually uses the access control list in paged pool. When the ACL processing is complete, the entire ACL is copied back into the file header on disk.

The ACL is returned to the user through a read attributes function, and it is set by WRITE_ATTRIB. The GET_FIB routine initially sets the FIB$L_ACL_STATUS field to success. This field gets its real value in the routines READ_ATTRIB and WRITE_ATTRIB. Writing to the ACL causes the FCB to be marked stale clusterwide, which forces the in-memory ACL to be rebuilt across the cluster.

The CHECK_DISMOUNT routine deletes any ACLs when it deallocates any FCBs associated with a device.

5.4.8 Dynamic Bad Block Processing

Bad block processing is not invoked explicitly by any user operations but happens as the result of normal file system processing in a way that is mostly transparent to the user. It is initiated when a hard I/O error occurs during file I/O, on either a read or a write function. The failing I/O is turned over to the file system, which flags the file as having a bad block but does nothing more at that time; dynamic bad block processing occurs when the file is deleted.

When this flagged file is deleted, it is sent to a special bad block scanner process instead of being directly deallocated. This process, created by the file system, asynchronously scans the file and accesses the blocks to search for the error.

5.4.8.1 Handling an I/O Error

When a read or write function on a file returns from a driver with an error status, the IOPOST routine clears the IRP$V_VIRTUAL bit in the IRP and queues it to the XQP. (Clearing the virtual bit distinguishes I/O error processing from requests for window turns and highwater mark processing, which arrive at the XQP with IRP$V_VIRTUAL set.) The XQP performs bad block processing if the error code in IRP$L_IOST1 is a parity, format, or datacheck error. Other error codes indicate problems other than media errors and do not cause bad block processing.

The MARKBAD_FCB routine in the RWVB module sets the bad block bit (FCB$V_BADBLK) in the indicated FCB. The routine SCAN_BADLOG is called to locate the block in BADLOG.SYS, the pending bad block log file. BADLOG.SYS is opened in secondary context and is searched for the LBN on which the I/O failed. If an entry is found, it is updated. If one is not found, it is created.

If the deaccess function sees that the bad block bit is set in the file header, it sets 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.

5.4.8.2 The Bad Block Scanner

Normally, when a file is deleted, the mapped blocks are returned to the storage bitmap. If the bad block 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, or the **bad block scanner**.

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 block. When the block that generated the error is found, it is exercised by being written and read three times to determine whether the error can be reproduced. If the block can be read and written satisfactorily all three times, then the file system continues to use the block. (In fact, most of these errors are not repeated, and the block is read and written satisfactorily.) However, if it cannot be read or written any one of those three times, the block is considered bad, and it is retired.

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 file to BADBLK.SYS).[1]

[1] Note that, logically, DSA disks contain no bad blocks. On a DSA disk, bad blocks are revectored in 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

For more information on bad block processing on DSA disks, see Section 2.5.3.3. 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 bad block 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.

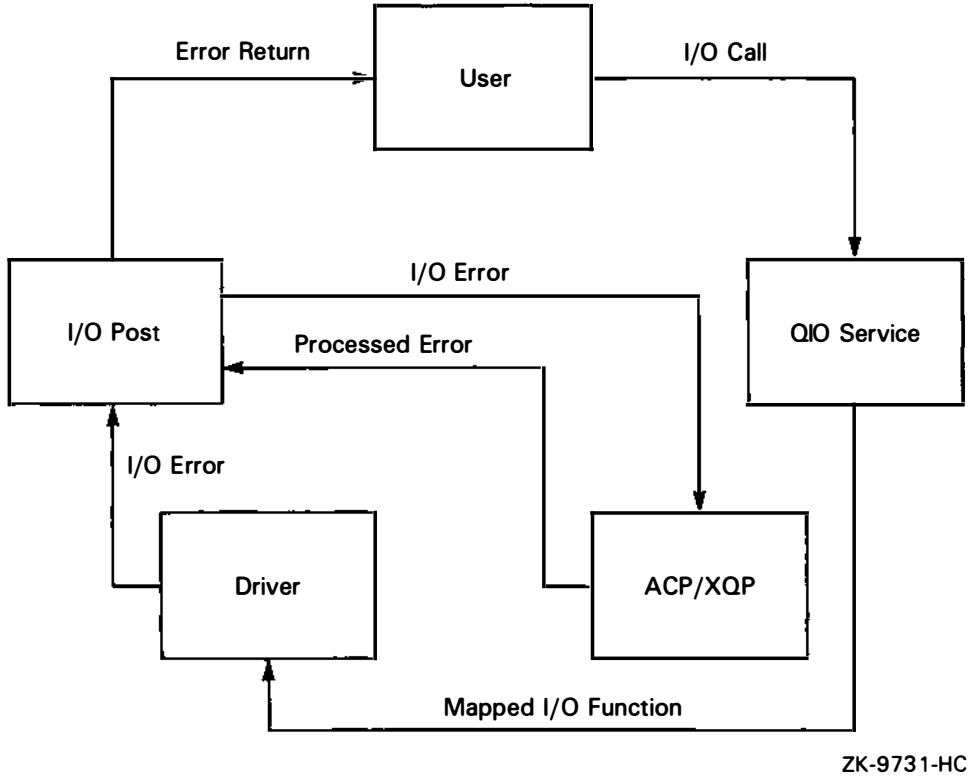
In secondary context, SCAN_BADLOG in the F11X module BADSCN is called to scan the pending bad block log and remove any existing BADLOG entries for these blocks. The bad block scanner will also check the BADLOG file for any references to the file when it is done.

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

Figure 5–2 shows a diagram of virtual I/O error handling.

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. In other words, if the file system finds a block with a forced error, the bad block processor runs when the block is deallocated. A DSA disk will try to read and write the block repeatedly; if the block cannot be read or written, it is revectored. So no bad blocks are entered in BADBLK.SYS for DSA disks.

Figure 5–2: Virtual I/O Error Handling

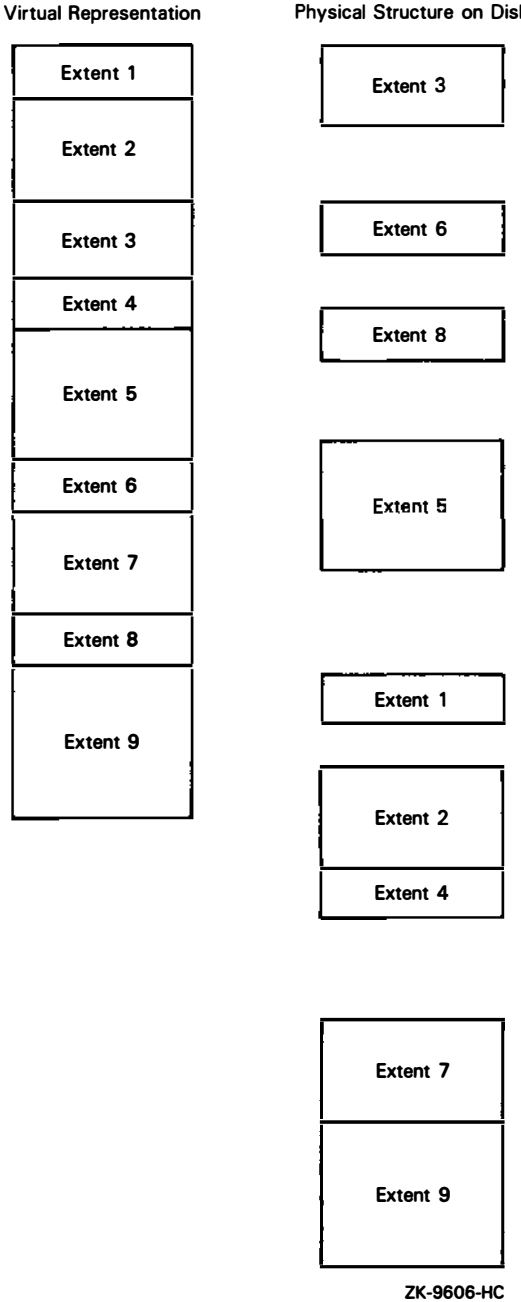


5.4.9 Window Handling

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 5–3 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 5–3: Virtual and Physical Representations of a File



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 LBN 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]
- The DCL command MOUNT/WINDOWS=n

A special type of window that maps the entire file is 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."

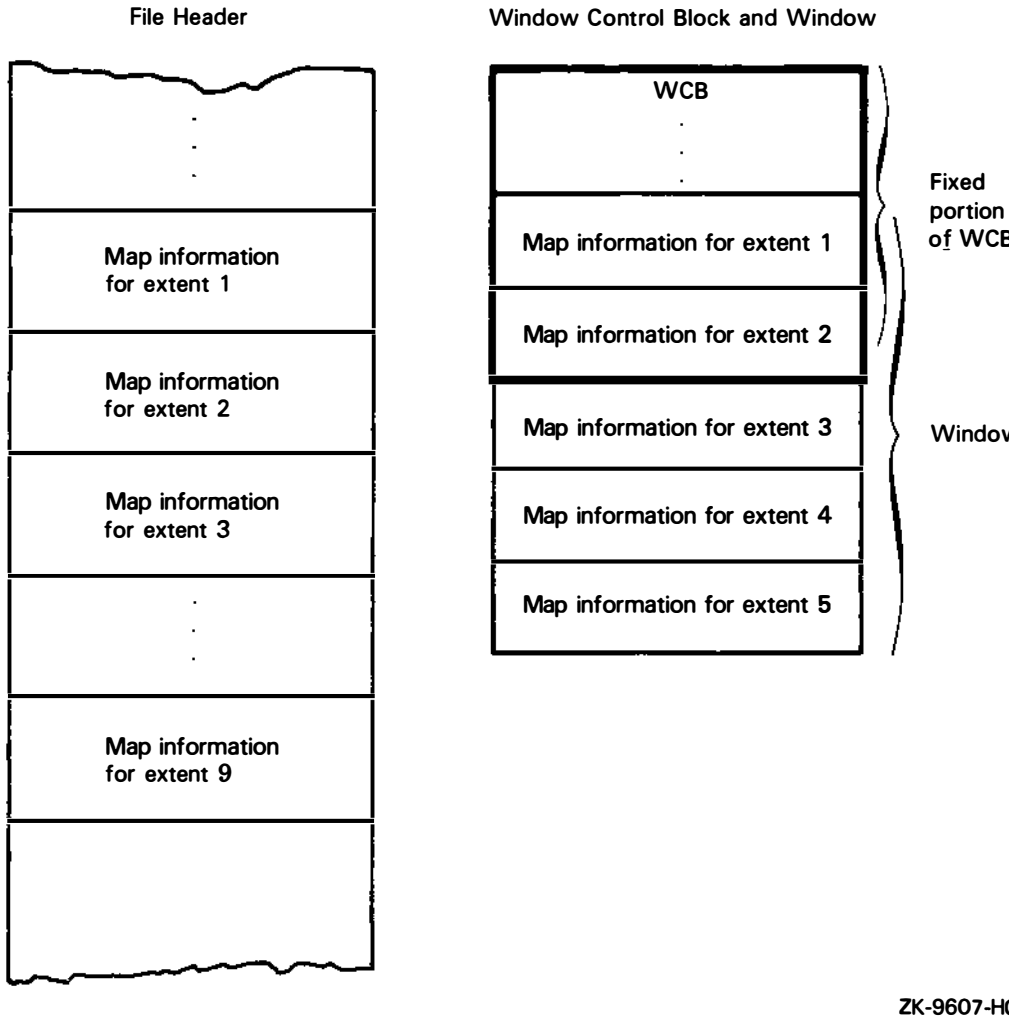
When a data transfer (a virtual read or write operation) is requested, a starting VBN and the size of the request in bytes is given. The file system then maps the VBN to an LBN, which is used to locate the file's blocks on disk.

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 header chain to find the file header that contains the desired retrieval pointer.

Figure 5–4 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 through 9 is needed, the XQP must turn the window.

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

Figure 5–4: Mapping a File with a Window Size of 5



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.

5.4.9.1 Mapping a Window

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.

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:

- 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.
- The window was previously complete, but the file was extended. In this case, new window pointers must be added to the last window segment, or a new WCB added.
- The window never completely mapped the file. In this case, the header 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:

- A successful mapping occurs because the current window contains the desired mapping information.
- A partial mapping occurs because the window contains the starting VBN, but it does not map contiguous extents.
- 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 precedes 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 volume containing the file.

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 checked for a segment is two. Although some DSA disks support longer transfers, this limitation only affects transfers greater than 65K blocks, which are extremely rare.

If the total transfer has been mapped contiguously, IOC$MAPVBLK performs the following actions:

- Returns the number of bytes mapped
- Returns the starting LBN
- Returns a status of SS$_NORMAL
- Allows interrupts
- Performs an RSB

However, if the transfer has not been completely mapped, the routine performs the following actions:

- Returns the number of unmapped bytes
- Returns the LBN of the first block mapped
- Returns status
- Performs an RSB

In both cases, if the file is on a volume set, the LBN field in the map pointer contains an RVN in bits <24:31>. This RVN is used to index into the RVT to fetch the UCB address for the volume containing the blocks mapped. The UCB address is returned to the caller.

After the file has been mapped, the IRP is queued to the driver's start I/O routine.

5.4.9.2 Turning a Window

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:

- The window must be turned to map a different portion of the file.
- The header contains pointers which may be added to the existing window after the existing window is truncated from the beginning.
- The desired VBN is less than the specified starting VBN and the starting VBN is greater than 1.
- 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 5–5 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 through 99) in the original window are discarded, and the new VBNs (VBNs 100 through 145) are copied to the window.

[1] Does not include cathedral windows.

Figure 5–5: Turning a Window Because of Complete Map Failure

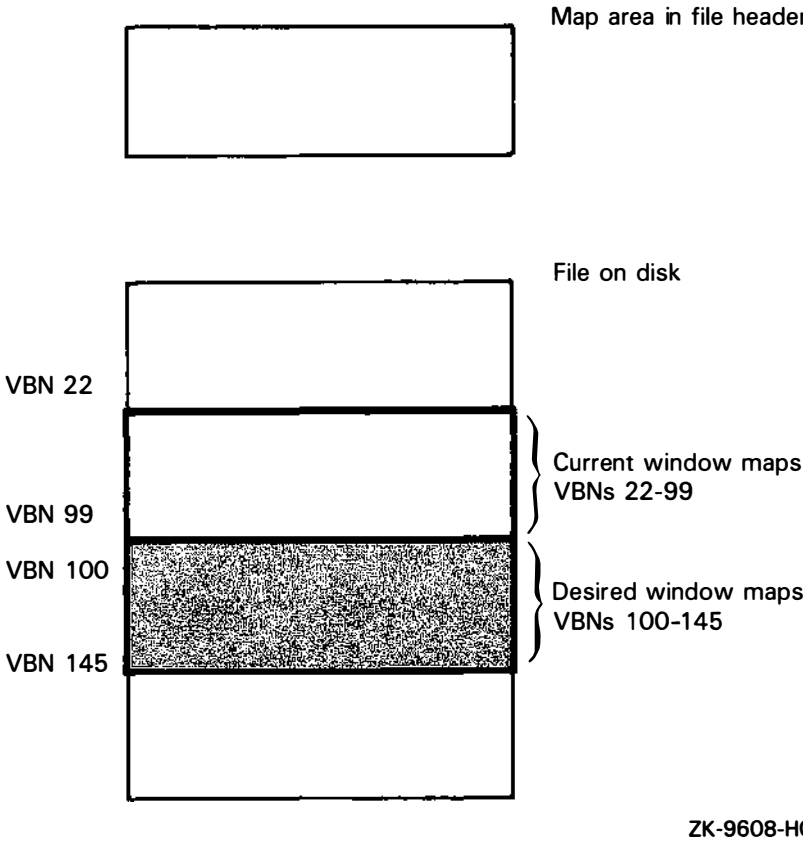


Figure 5–6 illustrates the second situation, where the header contains pointers that may be added to the existing window after the existing window is truncated from the beginning (or the top).

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 5–6: Turning a Window to Map Additional Pointers

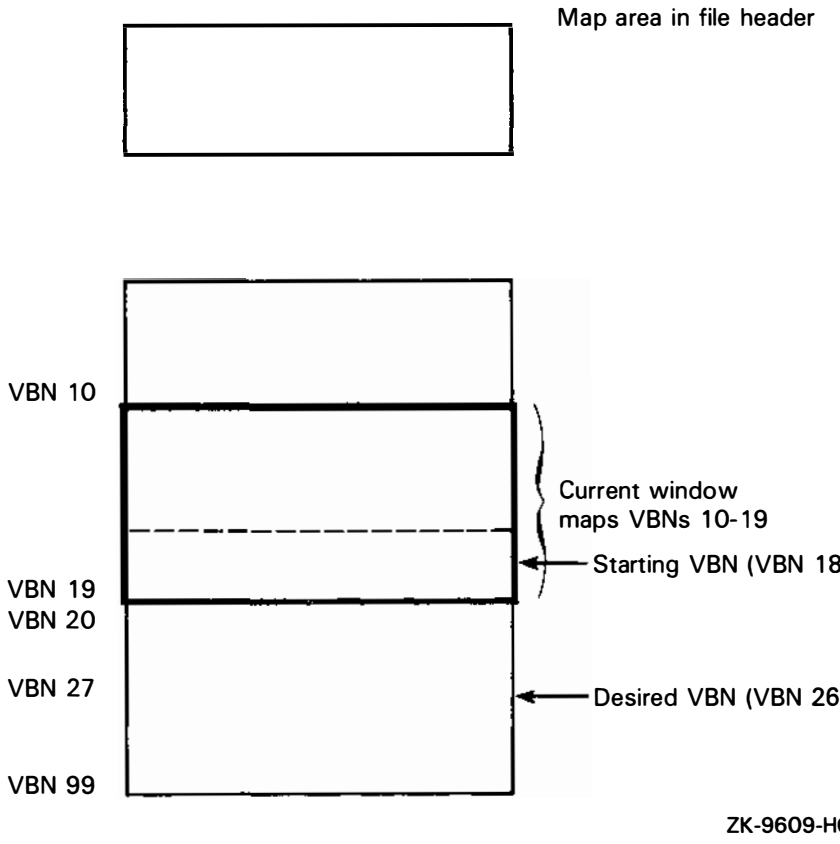


Figure 5–7 shows how the existing window is truncated from the top, or beginning, of the window. The pointer containing the starting VBN (VBN 18) was part of the old window, but it becomes the beginning of the new window. The new window also includes the desired VBN (VBN 26).

Figure 5–7: Truncating an Existing Window

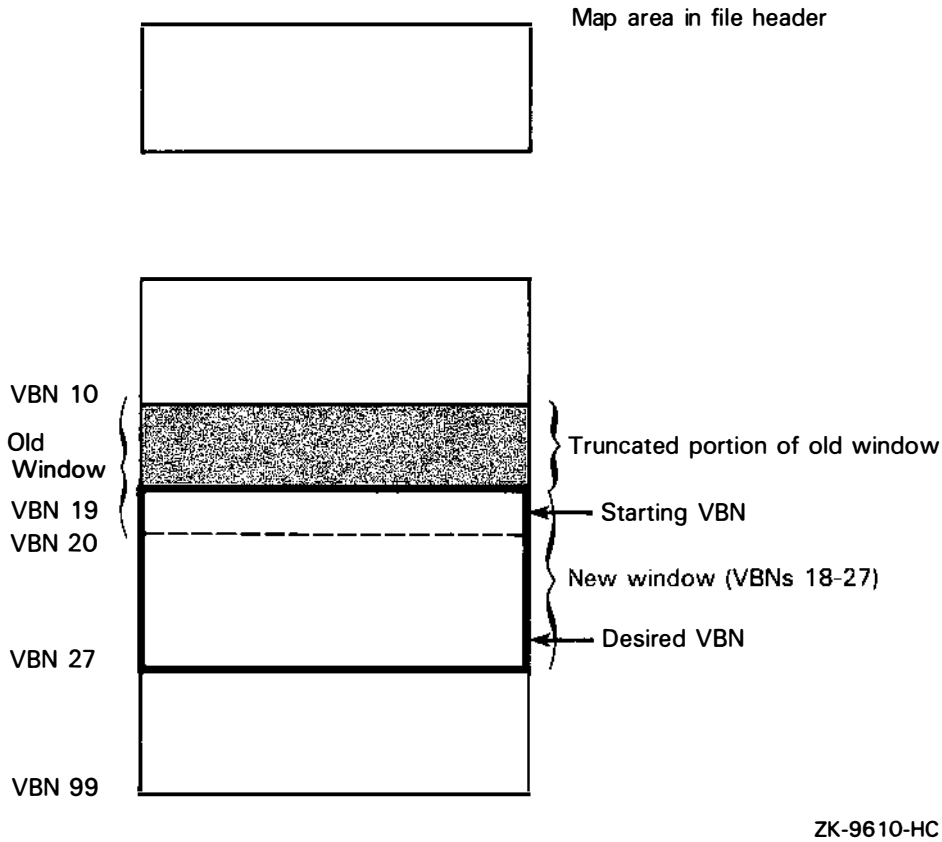


Figure 5–8 illustrates the third situation, where the desired VBN is less than the specified starting VBN and the starting VBN is greater than 1. This situation occurs when a file is extended and a new file header (an extension header) is created.

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

Figure 5–8: Turning a Window to Map a Previous Header

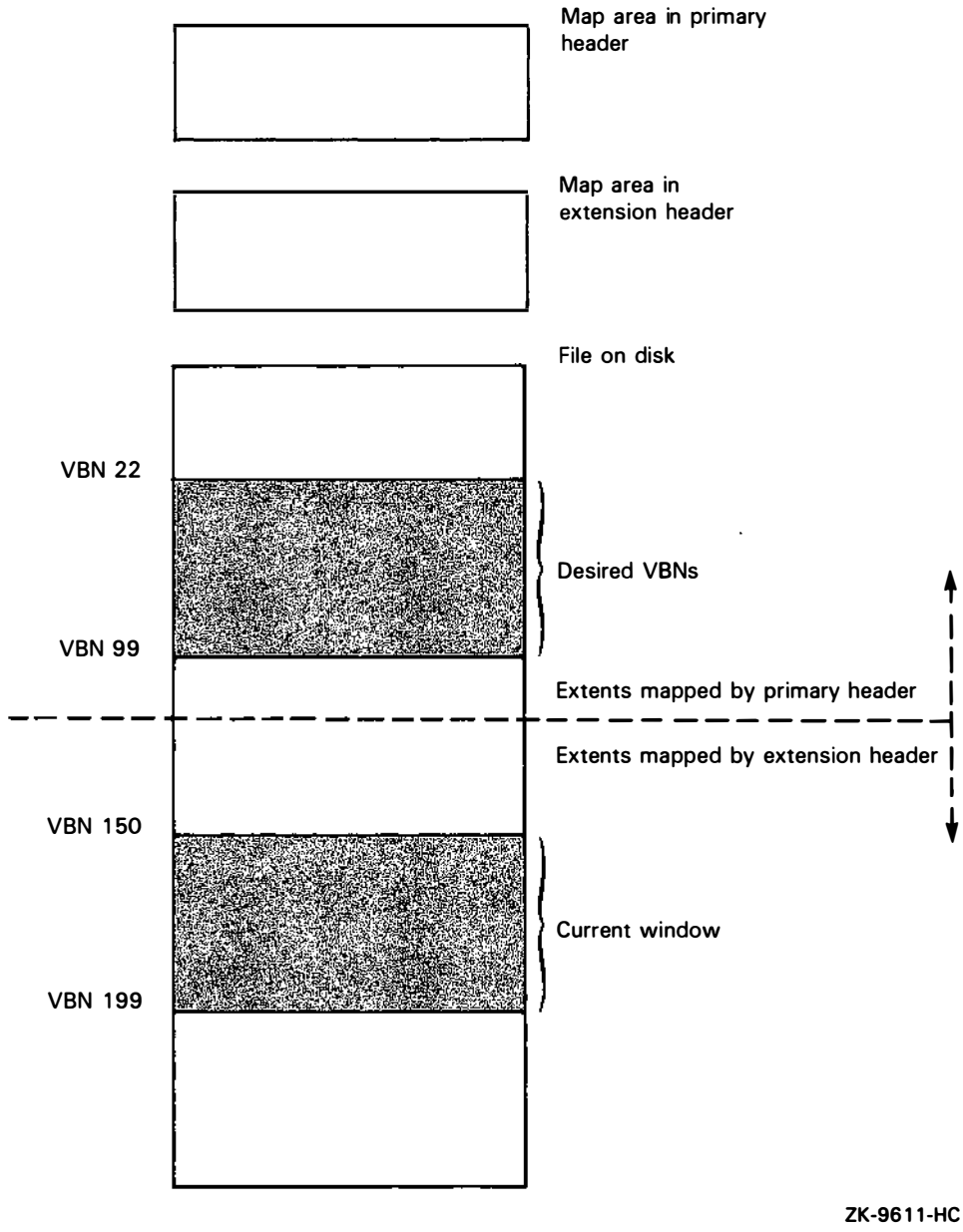


Figure 5–9 illustrates the fourth situation, where the window already maps a portion of the header and only the new pointers (which may include a partial map pointer if two contiguous extents were collapsed into one map pointer in the header) have to be mapped.

In this example, VBNs 1 through 100 are mapped by a single contiguous extent, and VBNs 101 through 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 5–9: Turning a Window to Map a Contiguous Extent

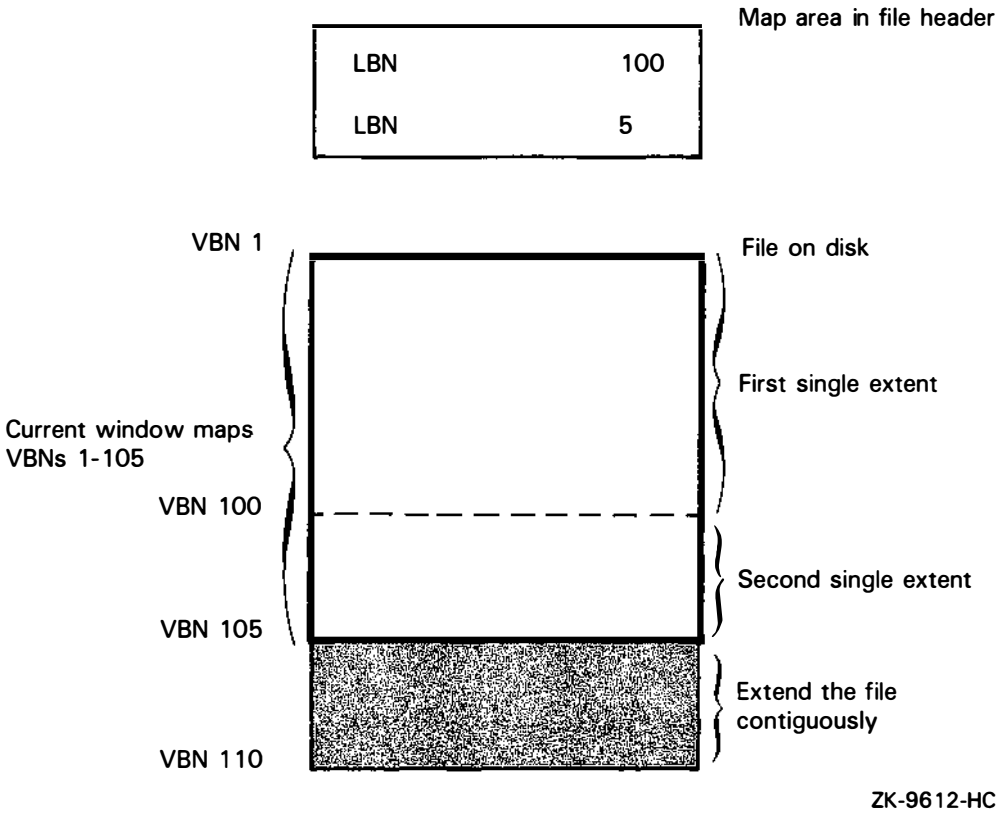
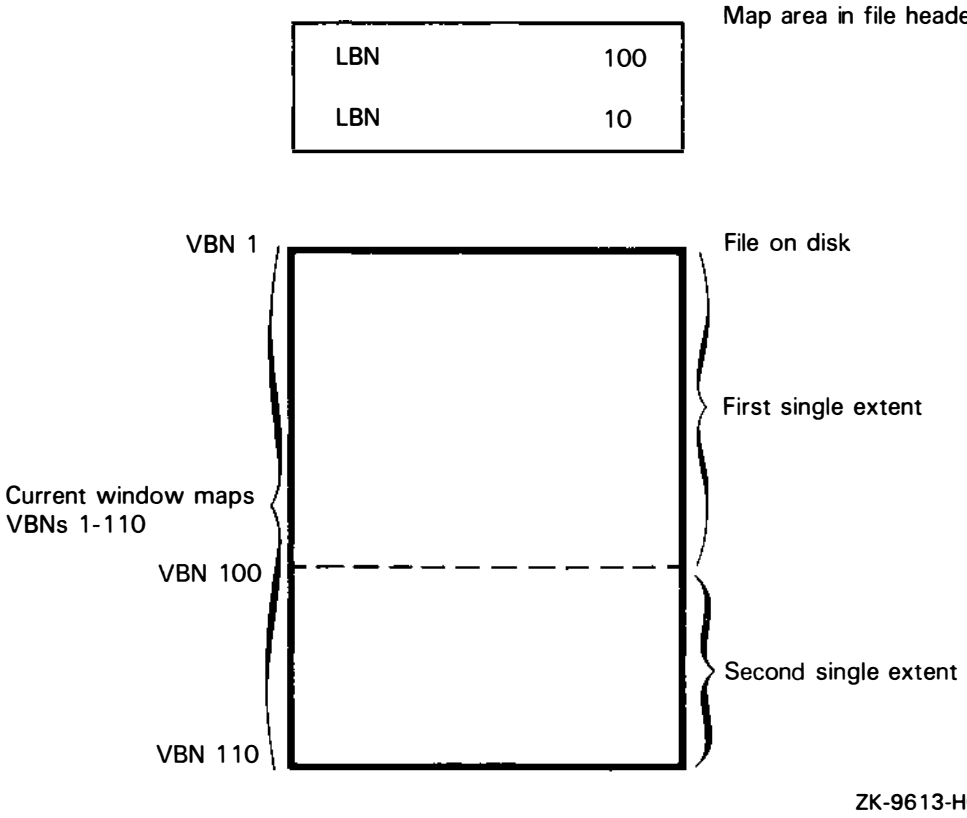


Figure 5–10 shows how the file system efficiently collapses, or combines, the contiguous extents into a single extent. The second pointer in the file header now reflects the addition of the new VBNs.

Figure 5–10: Collapsing the Contiguous Extents



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.

Once the block has been mapped, the IRP is requeued to the driver for which it was originally intended. 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 mapped at all, 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$CVTLOGPHY routine is called to convert a logical block to a physical block. After this conversion, the EXE$INSIOQ routine is called to queue the IRP back to the driver.

5.5 ACP Functions and Buffer Caching

The file "subsystem" uses or consumes resources external to itself as it does work for the user or the system. There are three major resources in VMS:

- Memory
- CPU
- I/O

Like any system, the file system must balance these resources in an adaptive fashion. A cache is nothing more than an in-memory resource management scheme.

File system operations involve at least two levels:

- Metadata, or information about the data, such as file headers, directory entries, and in-memory data structures
- Data, such as user blocks

Table 5–1 shows the I/O operations that the file system uses to create a file if caching is not available.

Table 5–1: Number of QIOs with Caching Disabled

Action	QIOs Used (Estimated)
Scan index file (that is, the bitmap) for a free header†	2
Read directory for placement†	3
Read storage bitmap to find free blocks†	4
Read quota block†	3
Write header	1
Write directory block	1
Write quota block	1
Total (estimated) number of QIOs	15

†Must be found before being allocated

This number represents the number of QIOs needed to create a file *before* data is written. That is, it represents only the number of QIOs needed to perform the housekeeping or overhead functions.

The balance is heavily against I/O, so caching is performed to balance I/O against memory. I/O is the limiting factor, and memory is inexpensive. Five special caches are as follows:

- Extent cache (bitmap cache)
- FID cache (file ID cache)
- Directory block cache
- Quota cache
- Directory index cache

Table 5–2 shows how many I/O operations the file system performs to do a create operation with a cache hit rate of 100%.

Table 5–2: Number of QIOs with Caching Enabled

Action	Cache Used	QIOs Used	QIOs Saved
Search in-memory bitmap and cache†	FID cache	0	2
Search cache by binary search†	Directory block cache and directory index cache	0	3
Search cache	Extent cache	0	4
Search cache	Quota cache	0	3
Write header (remove entry from cache)	FID cache	1	0
Update caches and write header	Directory block cache and directory index cache	1	0
Update (write-back) cache	Quota cache	0	1‡
Total number of QIOs		2	13

†Must be found before being allocated

‡Deferred

The performance gain with caching provides speed and efficiency for the price of some memory and CPU usage.

The file system has resources it must manage, and these resources have a price:

- Directory blocks
- Directory lookups (searches)
- Headers (file IDs)
- Quota entries
- Data blocks (bitmap)

So it makes sense to cache them, just as it makes sense to cache pages in the memory management subsystem (free page list and modified page list). In addition, the caches are coordinated across the cluster.

With a VAXcluster, caching is more important because there is more contention on each disk. It is more complex because the distributed or shared nature of a VAXcluster system necessarily requires more synchronization. However, there are more advantages.

There are two schemes that promote resource sharing within a VAXcluster system:

- Competition (highest demand is next)
- Sharing, or coordinated payback

Table 5–3 shows the types of file system caches and identifies the type of resource sharing they provide.

Table 5–3: Caches and Resource Sharing

Cache	Type of Resource Sharing
Quota	Competitive. Shifts as needed from node to node.
Extent	Shared cache with coordinated payback. Resource is shared or divided among the participants. If the usage on one node is very great, the coordinated payback is initiated, after which demand is lower. The cache flush blocking AST is used.
	Created on delete and truncate, debited on create and extend. Flushed by AST and DISMOUNT, and populated by MOUNT.
Directory index	Shared with coordinated invalidation.
Directory block	Shared with coordinated invalidation.
Bitmap block	Shared with coordinated invalidation.
Header block	Shared with coordinated invalidation.
File ID	Shared cache with coordinated payback. Resource is shared or divided among the participants. If the usage on one node is very great, the coordinated payback is initiated, after which demand is lower. The cache flush blocking AST is used.
	Created on delete and truncate, debited on create and extend. Flushed by AST and DISMOUNT, and populated by MOUNT.

Chapter 6

The XQP and I/O Processing

Dispatch is the soul of business.
Philip Dormer Stanhope, Earl of Chesterfield

A mighty maze! but not without a plan.
Alexander Pope

Outline

6.1 Introduction

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:

- Processes a nontransfer request (for example, a file access)
- Handles bad blocks found in the course of performing an I/O operation
- Processes 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 into and out of the XQP itself. 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

6.2 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.

In addition, when the XQP initializes the impure area in the SYSINIT process (the first process to execute in the system), it creates a permanent mailbox named ACP$BADBLOCK_MBX to communicate with the bad block processor.

6.2.1 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 and those portions of the impure area and the XQP code that may be referenced at an elevated IPL (any IPL greater than 2). In other words, 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 6.5.3.

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

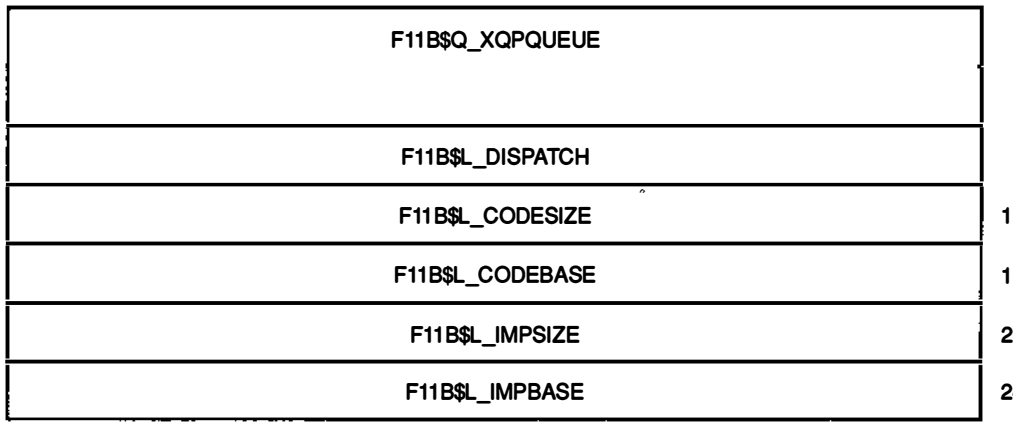


Table 6–1 describes the contents of the F11BXQP structure.

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	Entry point to the XQP for first-level request dispatching. This longword is a pointer to the DISPATCH routine in the DISPATCH module.
F11B$L_CODESIZE	Size of XQP code in bytes.
F11B$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 contains 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. All variables in the context area are defined as offsets to the base register. Register R10 is the base register for the XQP impure area, and it is initialized to the address labeled 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

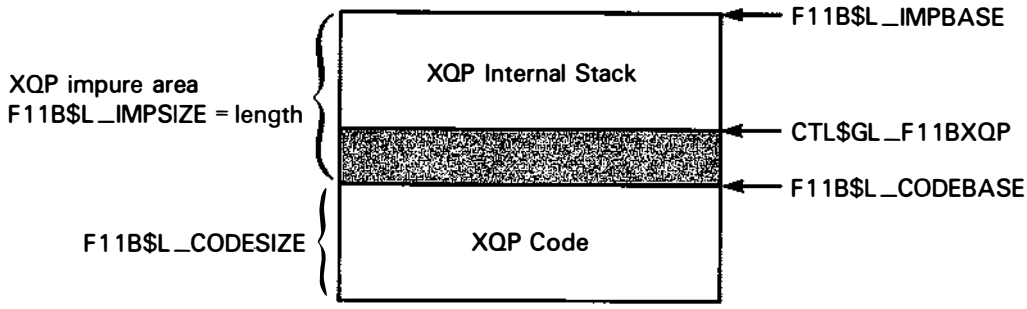


Figure 6–3 shows a further expansion of the XQP impure area. The impure storage area is delimited by the symbols STORAGE_START and STORAGE_END. The symbol L_DATA_START also points to the beginning of this area.

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 an 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 re-enterable 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 6.5.2.

Figure 6–3: Format of the Impure Area

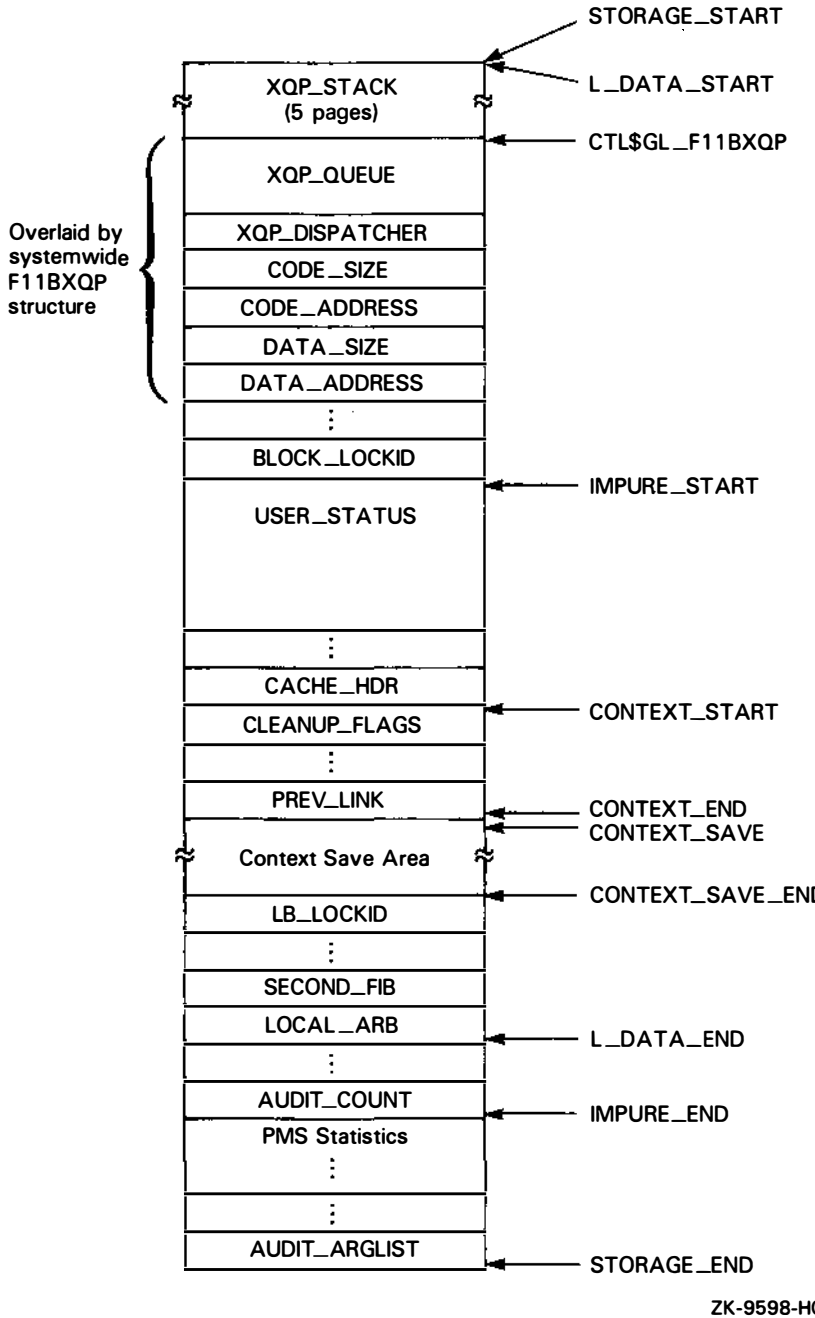


Table 6–2 lists all the symbols of the XQP impure area, their size, and a short description of each.

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$L_CODESIZE field.
CODE_ADDRESS	Longword	Base address of the XQP code. This cell corresponds to the F11B$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.

(continued on next page)

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

Impure Symbol	Size	Description
IO_CCB	Longword	Address of the channel control block of IO_CHANNEL, initialized by INIT_FCP. This cell locates the CCB, which in turn locates the UCB. That is, the structure represented by the construct IO_CCB[CCB$L_UCB] is essentially a volatile UCB pointer. For the duration of the current operation, IO_CCB[CCB$L_UCB] is set to CURRENT_UCB by GET_REQUEST and to the new UCB by SWITCH_VOLUME. It also locates the correct UCB while the least recently used (LRU) buffers in a cache are written to disk by WRITE_BLOCK.
IO_CHANNEL	Longword	Channel number of the channel pointed to by IO_CCB. This channel is used for all I/O issued by the XQP.
BLOCK_LOCKID	Longword	Lock ID of the activity-blocking lock held by this process. See Section 8.5.1 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.

(continued on next page)

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 zeroes it. For a contiguous extend function, 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 6.6.2 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.

(continued on next page)

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 nonzero 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.

(continued on next page)

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 Section 4.2.6 for more information.
BFR_CREDITS	4 words	Count of buffers credited to the process.
BFRS_USED	4 words	Count of buffers actually in-process.
CACHE_HDR	Longword	Address of the buffer cache header, set by GET_REQD_BFR_CREDITS.
CONTEXT_START	0	Label marking the beginning of the re-enterable 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 zeroes FILE_HEADER when the new header is written.
PRIMARY_FCB	Longword	Address of primary file FCB. This cell is set by the 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.

(continued on next page)

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 Section 7.7 for more information.
PRIM_LCKINDX	Longword	Primary file lock basis index.
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 the secondary context area.
CONTEXT_SAVE	54 bytes	Context save area. This area, currently 54 bytes, is where the primary context is saved when a secondary operation is performed.
CONTEXT_SAVE_END	0	Label marking the end of the context save area.

(continued on next page)

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

Impure Symbol	Size	Description
LB_LOCKID	5 longwords	Serialization lock IDs. This length of this cell is determined by the LB_NUM literal, which is the length of the lock basis vectors (representing the number of serialization lock blocks). Its value is currently 5; that is, the XQP is capable of holding up to 5 serialization locks at once, and one of those is the volume allocation lock. For more information on serialization locks, see Section 7.3.
LB_BASIS	5 longwords	This cell contains the file number, which, when combined with the appropriate lock name prefix, forms the resource name for the lock.
LB_HDRSEQ	5 longwords	File header cache sequence number. LB_HDRSEQ is incremented as blocks under the related lock are updated. For more information, see Section 8.6.1.
LB_DATASEQ	5 longwords	File data block cache sequence number. LB_DATASEQ is incremented as blocks under the related lock are updated. For more information, see Section 8.6.1.
LB_OLDHDRSEQ	5 longwords	Saved file header cache sequence number. This field contains the value of LB_HDRSEQ as it was read from the value block when the lock was taken. LB_OLDHDRSEQ is not incremented as blocks under the related lock are updated. Its value is used at the end of the RELEASE_LOCKBASIS routine to determine if each BFRD found under the lock in question was valid at the time the lock was taken.

(continued on next page)

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

Impure Symbol	Size	Description
LB_OLDDATASEQ	5 longwords	Saved file data block cache sequence number. This field contains the value of LB_DATASEQ as it was read from the value block when the lock was taken. LB_OLDDATASEQ is not incremented as blocks under the related lock are updated. Its value is used at the end of the RELEASE_LOCKBASIS routine to determine if each BFRD found under the lock in question was valid at the time the lock was taken.
QUOTA_DATASEQ	Longword	Current value of the sequence number for a quota file block.
QUOTA_OLDDATASEQ	Longword	Saved value of the sequence number for a quota file block.
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. See Section 7.7 for more information.
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.

(continued on next page)

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

Impure Symbol	Size	Description
PREV_VERSION	Longword	Version number of a directory entry being removed as a result of a create operation. It is used to restore the entry if the create operation fails.
PREV_NAME	80+1 bytes	Name string of a directory entry being removed as a result of a create operation. It is used to restore the entry if the create operation fails.
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	52 bytes	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	32 bytes	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.

(continued on next page)

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

Impure Symbol	Size	Description
IMPURE_END	0	Label marking the end of the impure data area zeroed at the start of each function.
MATCHING_ACE	512 bytes	Matching access control entry (ACE) storage, set by CHECK_PROTECT to the ACE which the access check matched, returnable via READ_ATTRIB.
NOTIFY_AST_ADDR	Longword	User notification AST routine. This field contains the user address at which to deliver file operation notification ASTs.
NOTIFY_NAME_LEN	Word	Length of file name. This field contains the length of the string built in NOTIFY_NAME_TEXT.
NOTIFY_NAME_TXT	89 bytes	Buffer to build a file operation notification message in response to events enabled with SET WATCH FILE. This unsupported utility allows types of calls to the XQP to be either displayed on a user's terminal or written to a selected batch log.
BLOCK_VCB	Longword	Locked volume. This field contains the VCB address of the volume a process has locked using the FIB$C_LOCK_VOL function.
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.

[1]In the absence of concealed device definition.

(continued on next page)

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

Impure Symbol	Size	Description
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 array is used to accumulate audit records.
STORAGE_END	0	Label marking the end of the impure storage area.

6.3 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).

6.3.1 I/O Request Packet

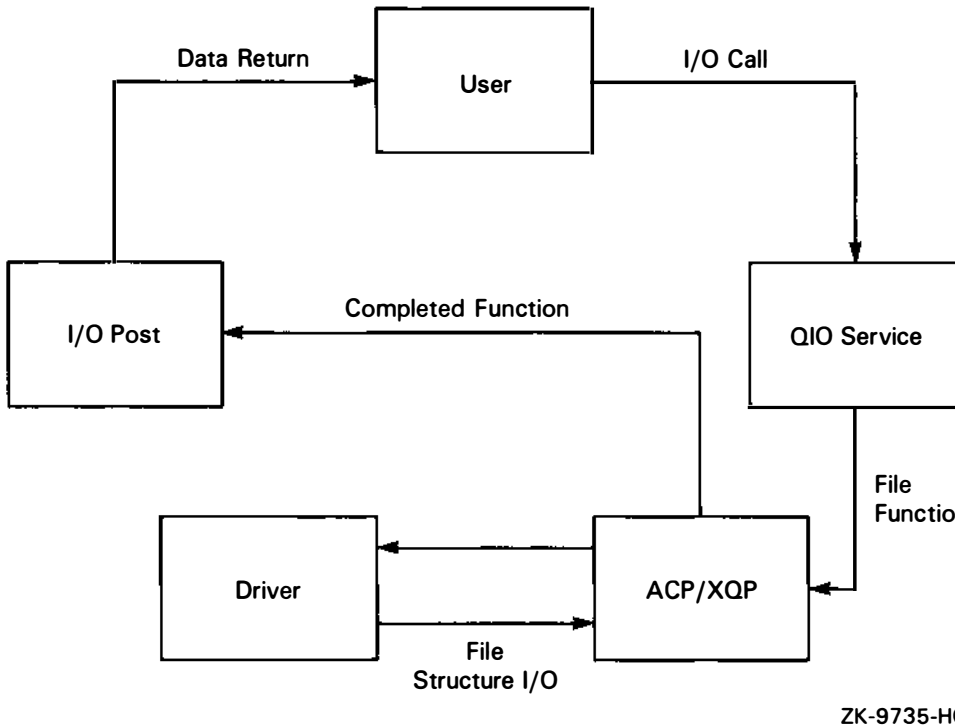
The **I/O request packet** (IRP) is the basic argument block passed to the file system for all functions. An IRP is a piece of nonpaged pool that describes the I/O request. When a process requests that I/O be performed, an IRP is constructed in a standard format.

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.

Figure 6–4 shows a diagram of a typical file system function.

Figure 6–4: File System Function



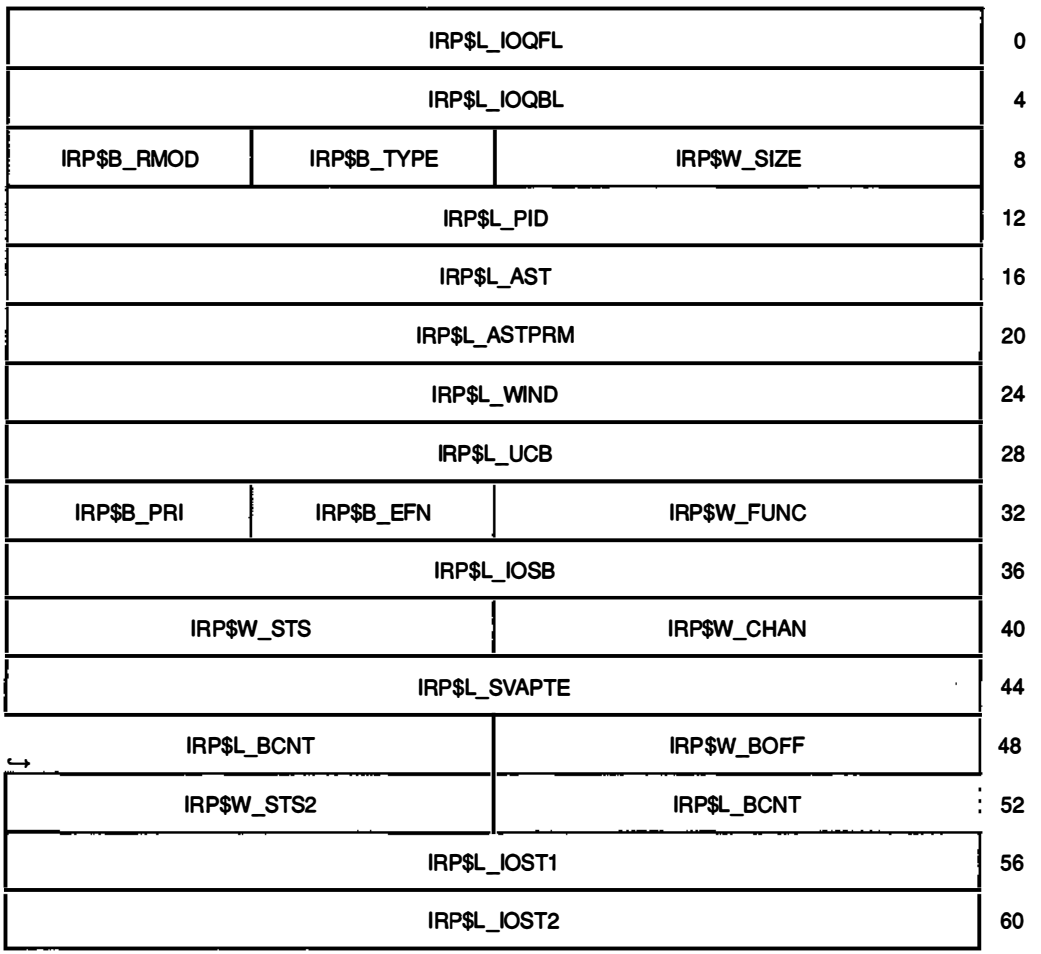
IRPs are a part of the I/O database. They are allocated from nonpaged pool. The IRP cannot be accessed by the user so the user cannot change the parameters after the $QIO system service has validated them and copied them into the IRP. Also, the driver can access queued packets without process context (and therefore the process page table). QIO fills in the first part of the packet from the device-independent parameters, of which there are six:

- Event flag number (EFN)
- Channel number
- I/O function code

- AST parameter
- AST routine address
- I/O status block (IOSB) address

The fields of the IRP are shown in Figure 6–5 and are described in Table 6–3. Note that the fields of the figure run right to left.

Figure 6–5: Format of the I/O Request Packet



(continued on next page)

Figure 6–5 (Cont.): Format of the I/O Request Packet

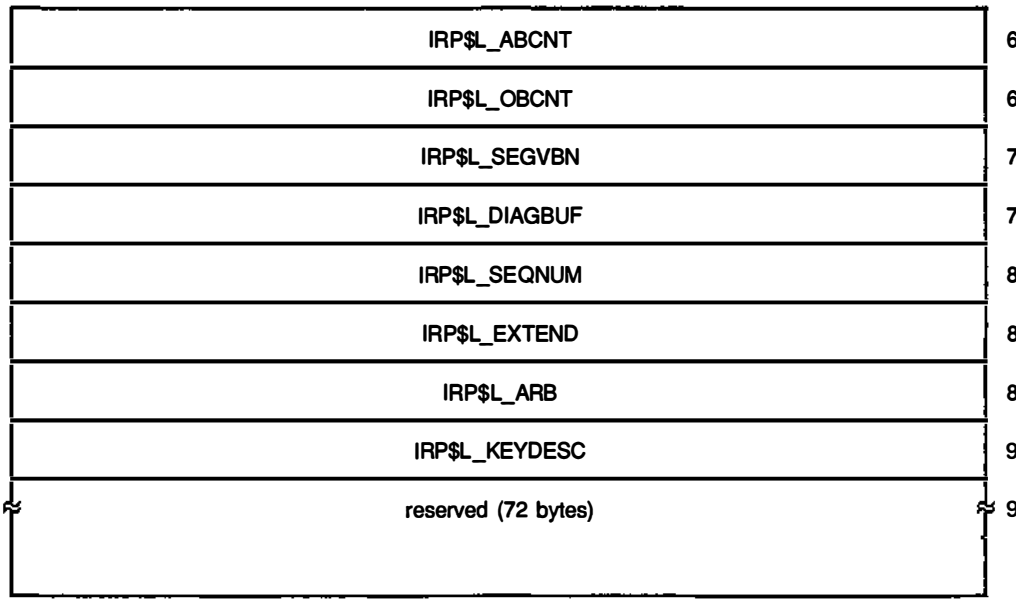


Table 6–3: Contents of the I/O Request Packet

Field Name	Description
IRP$L_IOQFL	I/O queue forward link. This field contains the address of the listhead of the queue for all systemwide pending I/O.
IRP$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$QIO routine writes the constant DYN$C_IRP into this field when the routine allocates and fills an IRP.

(continued on next page)

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

Field Name	Description
IRP$B_RMOD	Access mode of request. IRP$B_RMOD contains information used by I/O postprocessing. It also contains the same bit fields as the ACB$B_RMOD field of an AST control block. The EXE$QIO routine obtains the processor access mode from the PSL and writes the value into this field. One field is defined within IRP$B_RMOD—IRP$V_MODE, which is the mode subfield. This field indicates the mode of the process at the time of the I/O request. 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 <0:1>.
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. During I/O postprocessing, the special kernel-mode AST routine queues a user mode AST to the requesting process if this field contains the address of an AST routine.
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. During I/O postprocessing, the special kernel-mode AST routine queues a user-mode AST if the IRP$L_AST field contains an address, and passes the value in IRP$L_ASTPRM to the AST routine as an argument.
IRP$L_WIND	Address of the window control block. This field contains the address of the WCB that describes the file being accessed in the I/O request. EXE$QIO writes this field if the I/O request refers to a file-structured device. The XQP (or ACP) reads this field.
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.

(continued on next page)

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

Field Name	Description
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. 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 by default. EXE$QIO writes this field, and the I/O postprocessing routine calls SCH$POSTEF to set this event flag when the I/O operation is complete.
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. The I/O postprocessing special kernel-mode AST routine writes two longwords of I/O status into the IOSB after the I/O operation is complete.
IRP$W_CHAN	Process I/O channel number. This field contains the index number of the process I/O channel for the I/O request. EXE$QIO writes this field.
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.

(continued on next page)

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

Field Name	Description

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.

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 complex buffered I/O, and long virtual I/O.

Complex buffered I/O and **chained buffered I/O** are used by the XQP. Complex buffered I/O is used for access and deaccess 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$V_PAGIO	Pager I/O function. This is bit 18.
IRP$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.

(continued on next page)

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

Field Name	Description	
	IRP$V_SWAPIO	Swapper I/O function. This is bit 22.
	IRP$V_DIAGBUF	Diagnostic buffer allocated. This is bit 23.
	IRP$V_PHYSIO	Physical I/O function. This is bit 24.
	IRP$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$V_FILACP	File ACP I/O. This is bit 28.
	IRP$V_MVIRP	Mount verification IRP function. This is bit 29.
	IRP$V_SRVIO	Server-type I/O. This bit indicates that mount verification is triggered on an error but the I/O is not stalled. This is bit 30.
	IRP$V_KEY	IRP$L_KEYDESC contains the address of a key used for encryption. This is bit 31.
IRP$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.	
	I/O postprocessing uses this field to deallocate the system buffer for a buffered-I/O transfer or to unlock pages locked for a direct-I/O transfer.	
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.	

(continued on next page)

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

Field Name	Description
IRP$L_BCNT	Byte count of the I/O transfer. FDT routines calculate the count value and write the field. IOC$INITIATE copies the low-order word of this field into UCB$W_BCNT before calling a device driver's start I/O routine. For a buffered I/O read function, I/O postprocessing uses IRP$L_BCNT to determine how many bytes of data to write to the user's buffer.
IRP$W_STS2	Second word of I/O request status. EXE$QIO initializes this field to 0. EXE$QIO, FDT routines, and driver fork processes modify this field according to the current status of the I/O request. The following bits are defined within IRP$W_STS2. These bits are adjacent and in order. IRP$V_START_PAST_HWM — I/O starts past the file highwater mark. IRP$V_END_PAST_HWM — I/O ends past the file highwater mark. IRP$V_ERASE — Erase I/O function. IRP$V_PART_HWM — Partial file highwater mark update. (The count of partially validated highwater mark operations is contained in the FCB$W_HWM_PARTIAL field.) IRP$V_LCKIO — Locked I/O request, as used by DECnet direct I/O.
IRP$L_IOST1	First I/O status longword. The I/O postprocessing routine copies the contents of this field, also called IRP$L_MEDIA, into the IOSB.
IRP$L_IOST2	Second I/O status longword. The contents of this field are also copied into the IOSB during I/O postprocessing. The low byte of this field is also called IRP$B_CARCON. It contains carriage control instructions to the driver.
IRP$L_ABCNT	Accumulated bytes transferred. IOC$IOPOST reads and writes this field after a partial virtual transfer.
IRP$L_OBCNT	Original transfer byte count. IOC$IOPOST reads this field to determine whether a virtual transfer is complete, or whether another I/O request is necessary to transfer the remaining bytes.

(continued on next page)

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

Field Name	Description
IRP$L_SEGVBN	Virtual block number of the current segment of a virtual I/O transfer. IOC$IOPOST writes this field 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. IOC$IOPOST reads this field. The IRP$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, which are set up as follows: ARB$Q_PRIV Quadword containing process privilege mask SPARE$L Reserved longword ARB$L_UIC Longword containing process UIC
IRP$L_KEYDESC	Address of encryption descriptor.

IRPs appear on a variety of queues. Those queued to a particular device driver, for example, are linked to the UCB$L_IOQFL and UCB$L_IOQBL listhead. All IRPs waiting for completion processing are queued to the global cell IOC$GL_PSFL.

An IRP may also be used as an AST control block. For more information, see Section 6.4.5.

6.3.2 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, the FDT is pointed to by the associated driver dispatch table. The FDT routines execute in process context, and they access process space (P0 and P1). The file system is primarily concerned with five major FDT functions:

- Access (and create)
- Deaccess
- Modify (and delete)
- Mount
- Read (and write)

All disk drivers that support the VMS file system are expected to use the FDT routines for file system functions 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 through 5) contain the function code.
- The remaining 10 bits (bits 6 through 15) contain the modifier code.

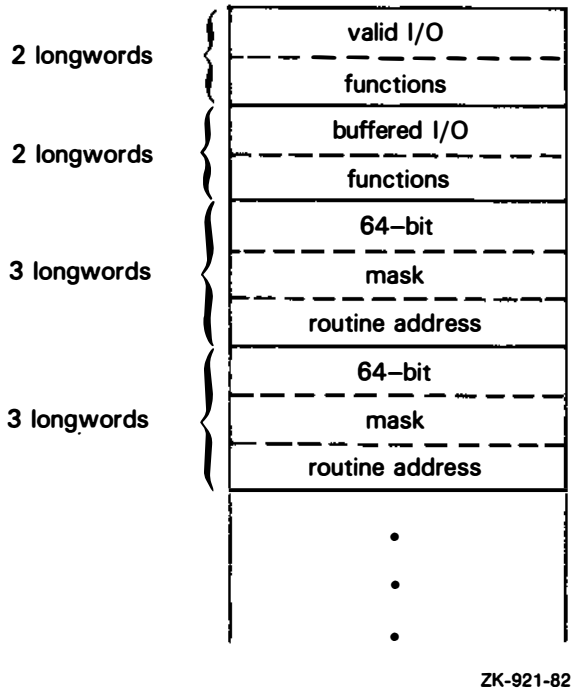
The 6-bit function code is used as a bit number into the 64-bit masks. If the bit number corresponding to the I/O function is set in the mask, QIO dispatches to the action routine.

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.

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 6–6 shows the format of a function decision table.

Figure 6–6: Format of a Function Decision Table



6.3.3 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.

6.4 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 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.

6.4.1 $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:

- Specifying the address of an AST routine to be executed when the I/O completes
- Waiting for the specified event flag to be set
- Using the $SYNCH system service to wait for the I/O status block (IOSB) to be completed and the event flag to be posted.

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; the driver defines FDT routines 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 to the channel.
- 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 information, 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 IPL$_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 functions on the channel. While an access or deaccess function is in progress, bit <0> of the WCB pointer is set to 1 to indicate that the channel is in transition. If the $QIO service finds the bit set, it goes into ASTWAIT state and retests the bit until the pending access or deaccess has completed and the bit is clear.

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 causes 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$_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.

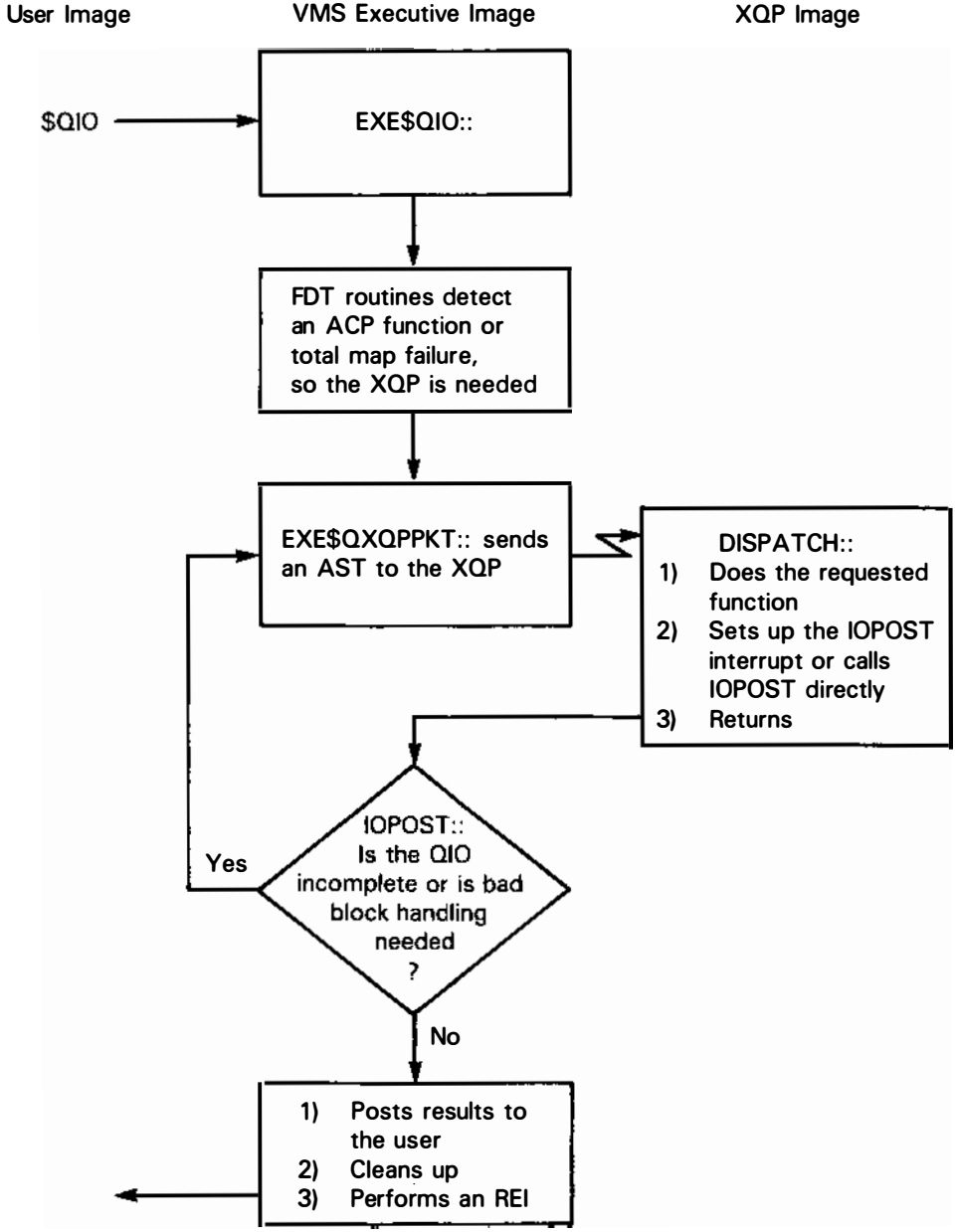
The IRP is then filled in; the function-independent parameters and process information are copied to the I/O packet. The I/O function code is validated against process privilege and device characteristics.

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 6–7 shows the user, VMS executive, and XQP images that are executed while the XQP processes an I/O request.

Figure 6–7: Images Used to Process an I/O Request



ZK-9599-HC

6.4.2 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.

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 are responsible for all device- and function-dependent processing. They interpret the device-dependent parameters of the $QIO argument list and translate them into fields in the IRP. The FDT routines are also responsible for proper handling of the IRP, ultimately by queuing it to a driver, to a file processor, or even to I/O postprocessing for immediate completion. $QIO continues to call FDT routines until one exits the $QIO service by executing a RET instruction.

The SYS module SYSACPFDT contains the FDT routines used to handle all processing related to the disk file system. For each IO$_xxx function, there is an ACP$xxx routine (such as ACP$ACCESS, ACP$CREATE, and ACP$READVBLK).

The virtual I/O functions fall into two major categories:

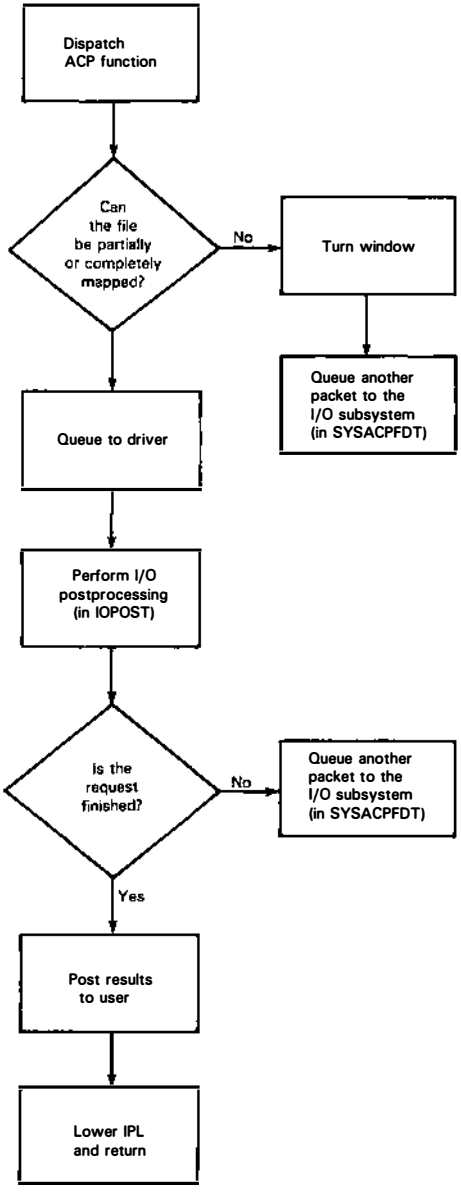
- **File functions** such as access, deaccess, create, and so on. These operations result in the queuing of a buffered IRP to the XQP for processing.

  If a **nontransfer request** (such as a deaccess function if the process was not the last writer, or an access to an already accessed file) was specified, an XQP packet is built instead. See Section 6.4.3 for more information on building the XQP I/O packet.

- **Transfer functions** such as read and write virtual. This type of operation is usually converted into a physical transfer operation by the FDT routine and then directly queued to the driver. The mapping data in the file's WCB is used to translate file virtual blocks into physical disk addresses. Only if this translation fails is a transfer request queued to the XQP.

Figure 6–8 shows the logic that determines the action the file system takes when a transfer request is initiated.

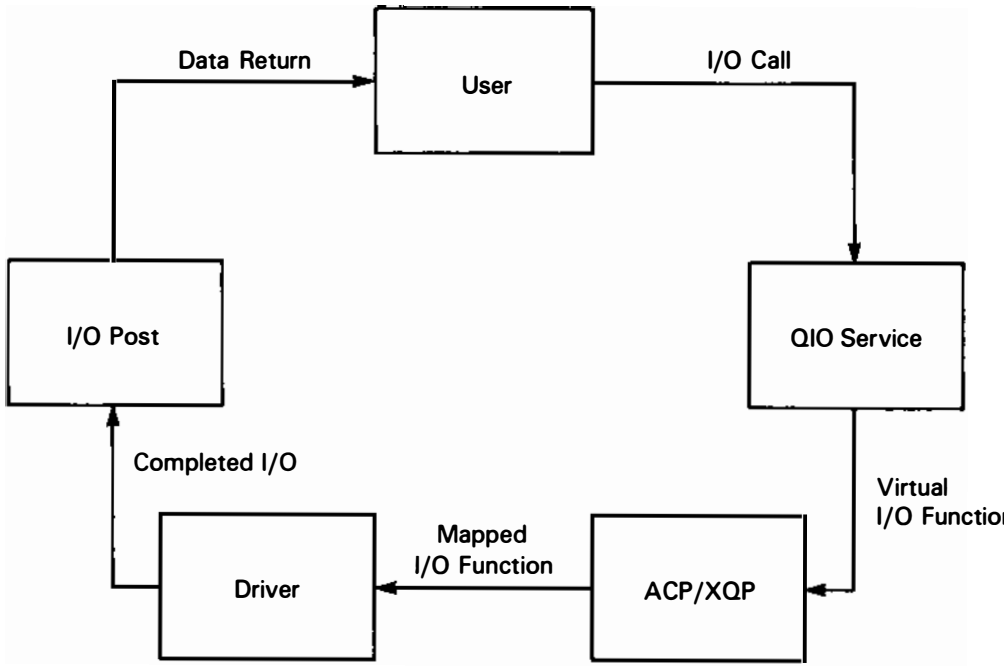
Figure 6–8: XQP Logic for an I/O Transfer Request



ZK-9600-HC

Figure 6–9 shows a diagram of a virtual transfer function.

Figure 6–9: Virtual Transfer Function

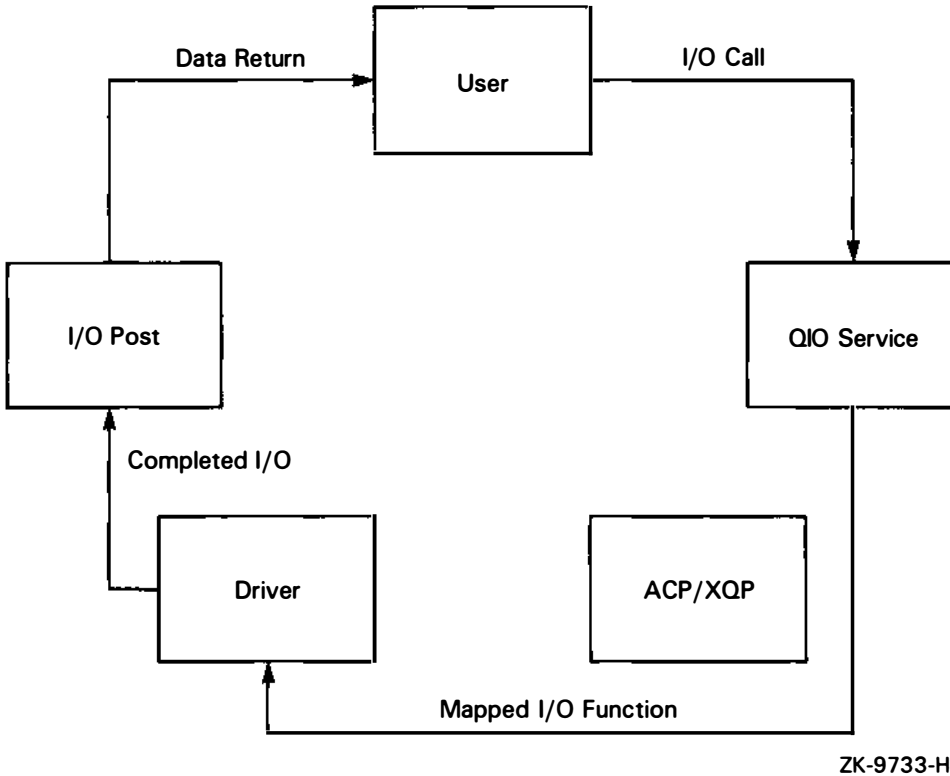


When a transfer request is processed, one of two cases exists:

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

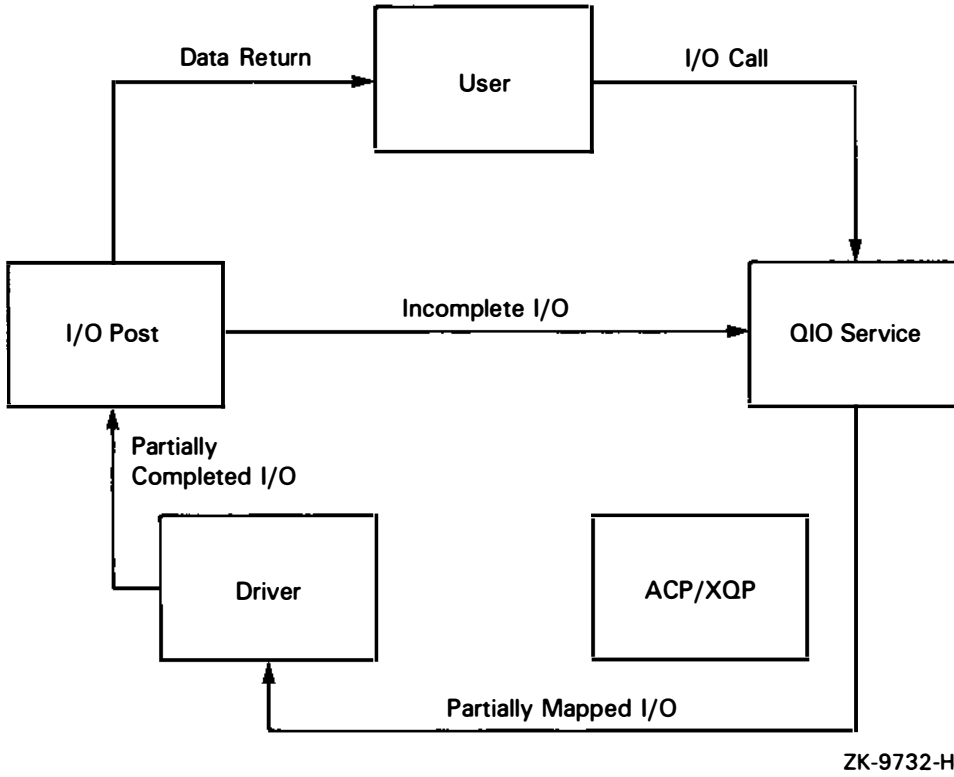
In the first case, the available information is queued to the driver, and the results are posted to the user. Figure 6–10 shows a diagram of a mapped virtual transfer function.

Figure 6–10: Mapped Virtual Transfer Function



If the virtual transfer function is fragmented, the file system handles that condition by cycling between the IOPOST routine and the driver, mapping each successive fragment with each loop through IOPOST. Figure 6–11 shows a diagram of a fragmented virtual transfer function.

Figure 6–11: Fragmented Virtual Transfer



If total mapping failure occurs, 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).

6.4.3 Building the XQP I/O Packet

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

The AIB has a 12-byte header. The first longword, the AIB$L_DESCRIPT field, points to a vector of buffer descriptors. Between the header and the buffer descriptors is a copy of the fixed portion of the ABD for the process. **ABD buffer descriptor**, or **ABD**, 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 6–16). 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 6.7.

The format of the XQP I/O buffer packet header is shown in Figure 6–12 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 6–12: Format of an XQP I/O Buffer Packet Header

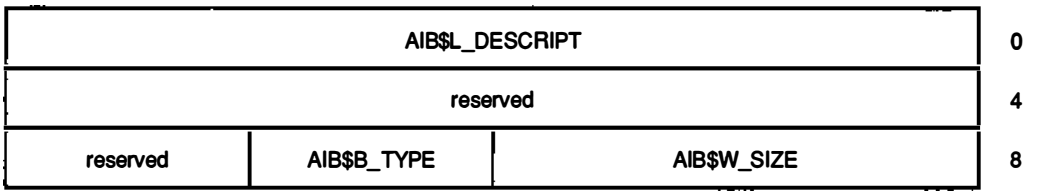


Table 6–4: Contents of an XQP I/O Buffer Packet Header

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 performs the following actions:

- Ensures that the buffer byte count quota has not been exceeded
- Checks the user's parameters to the QIO (such as the FIB) to ensure that they can be accessed
- Allocates the buffer
- Inserts the buffer descriptors (ABDs) for each user parameter in the XQP packet

A **complex buffer** is a set of pointers to a collection of ACP buffer descriptors (ABDs). It 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.

Each ACP 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 virtual read or write function, IRP$V_COMPLX is clear, so the above description does not apply).

The fields of a ACP buffer descriptor are shown below in Figure 6–13 and are described in Table 6–5.

Figure 6–13: Format of an ACP Buffer Descriptor

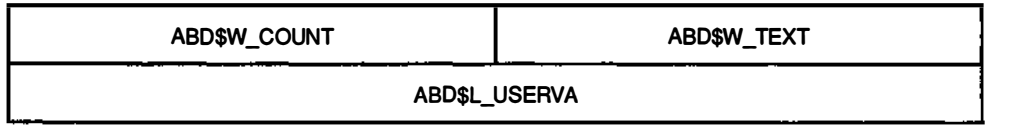


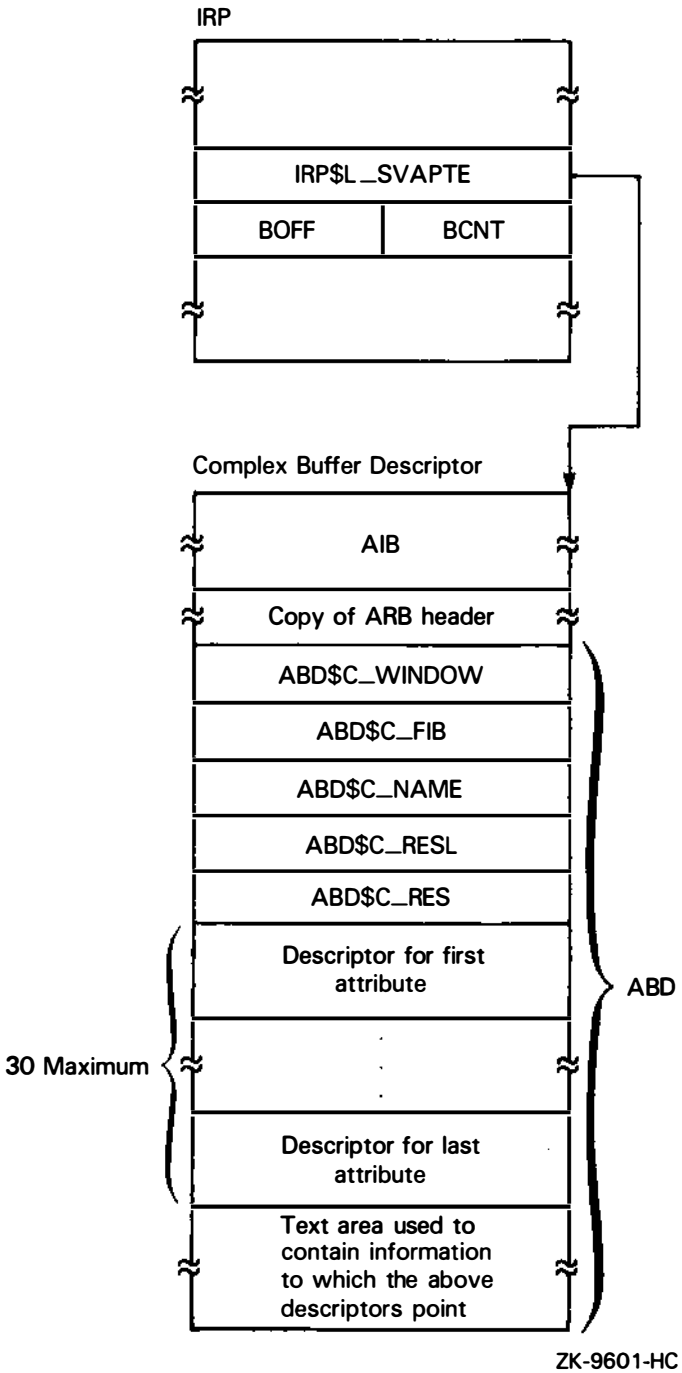
Table 6–5: Contents of an ACP Buffer Descriptor

Field Name	Description
ABD$W_TEXT	Word offset to the data text in the text area. Figure 6–15 shows the format of a single data text entry.
ABD$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 SVAPTE field of the IRP 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.

Figure 6–14 shows how the complex buffer descriptor is constructed.

Figure 6–14: Locating the Complex Buffer Descriptors



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

- **ABD$C_WINDOW**

  The first descriptor 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 zeroes 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.

- **ABD$C_FIB**

  The second descriptor 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.

- **ABD$C_NAME**

  The third descriptor contains the file name buffer. It is passed as the input to the PARSE_NAME routine from the enter and find functions to parse the user's file name into the internal name block. The COPY_NAME routine (called from the create and find functions for spooled devices) copies the file name buffer into the result string buffer. It also sets the result string length buffer value.

  The file name string travels only in one direction: from the user to the XQP. Its counterpart, the result name string, is what is sent back to the user from the XQP. For efficiency, the IO_DONE routine clears the file name 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. For a deaccess function on a spooled device, FDT processing places the user name and account in the file name string descriptor to be sent to the job controller.

- **ABD$C_RESL** and **ABD$C_RES**

  The fourth and fifth descriptors 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 processes a wildcard search. Quota file operations call the RET_QENTRY routine 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 function. The IO_DONE routine sets the IRP$L_BCNT field during nonread 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 from create and modify functions, scans the user's attribute list for placement data and copies it into standard format in the FIB.

The format of a data text entry is shown in Figure 6–15. 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 6–15: Format of a Data Text Entry

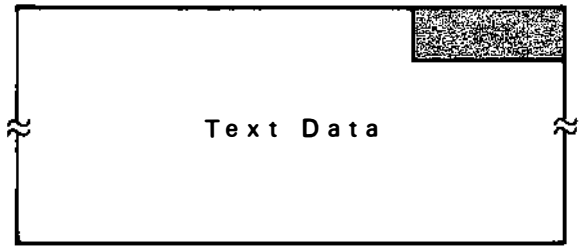
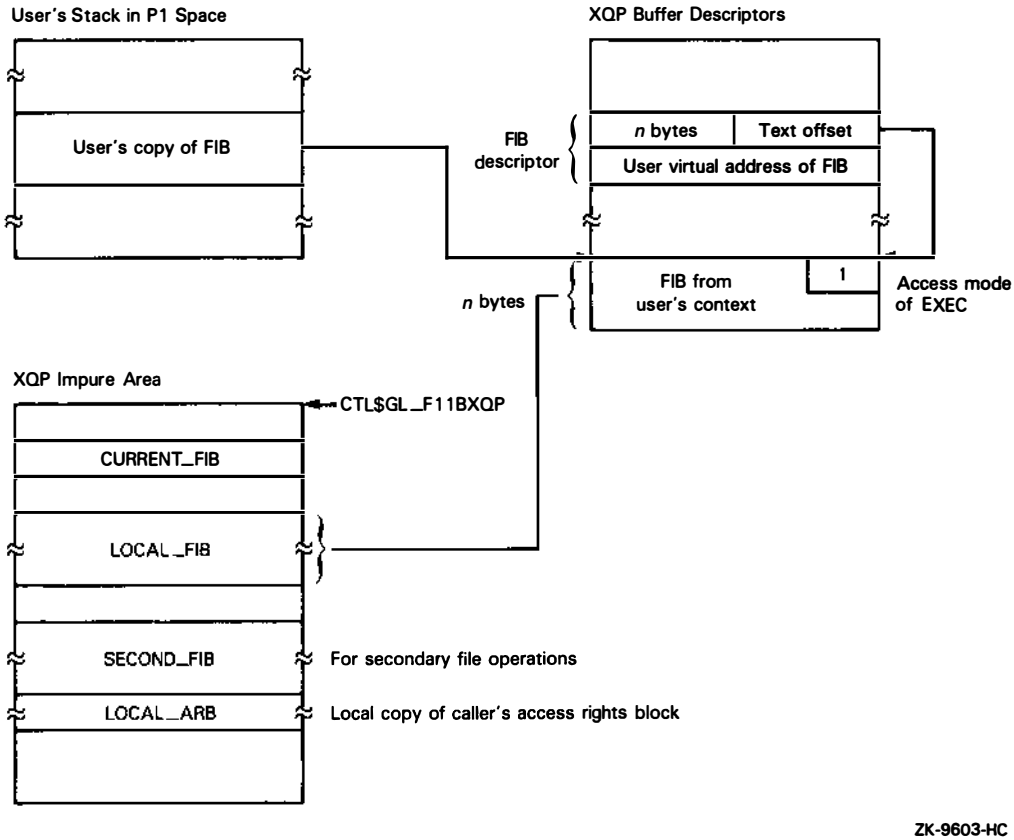


Figure 6–16 shows how user information is copied from user context to the XQP. In this case, a FIB is copied from the user's stack in P1 space into the data text portion of the ABD. A FIB descriptor is created to describe and locate the data.

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.

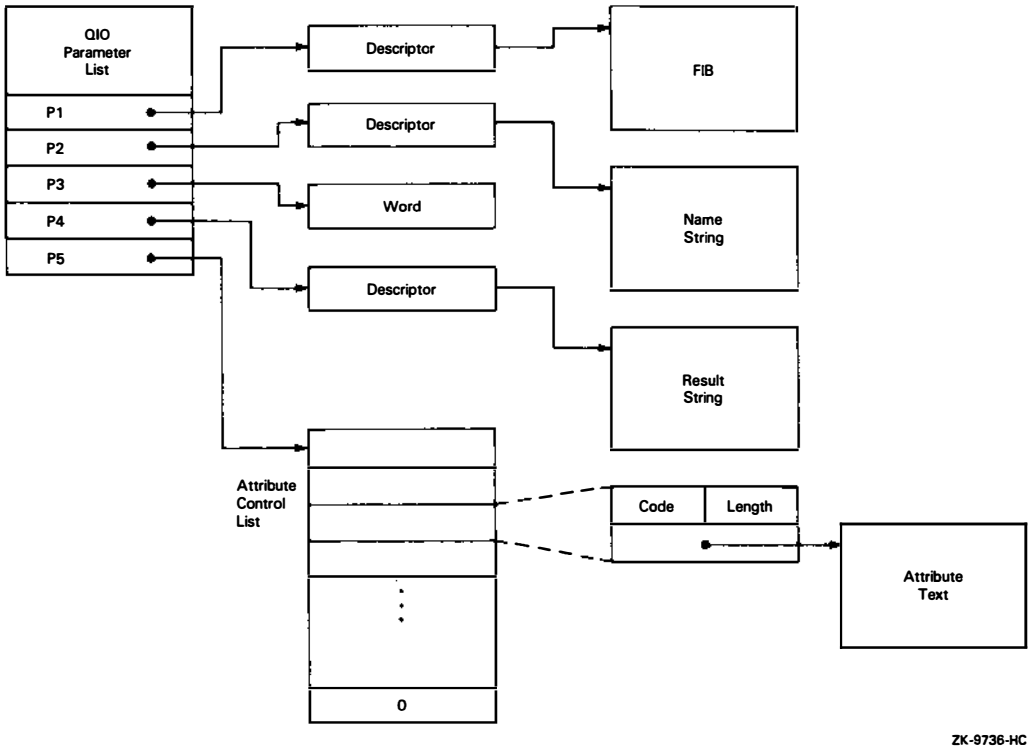
Figure 6–16: Passing User Information to the XQP



The number of descriptors is placed in the IRP$L_BCNT field, and the number of bytes charged to the buffer byte count quota is written into the IRP$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.

Figure 6–17 shows the relationship of all the components of the XQP–QIO interface.

Figure 6–17: XQP–QIO Interface



6.4.4 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 one of 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$V_DMT bit is set in the UCB$L_DEVCHAR 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.

The channel cannot be closed until outstanding QIOs are completed. If the function is IO$_DEACCESS and the I/O count in the channel is nonzero, the address of the IRP is saved in CCB$L_DIRP and the $QIO service returns to the caller. The pending deaccess will be queued to the file system when other outstanding I/Os have completed.

6.4.5 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$WAKACP in the SYS module IOCIOPOST via a special kernel AST. The special kernel AST ensures that the IRP will not be invalidated by process deletion between the time IOPOST is exited and the time a normal kernel AST could be delivered to the process.
- EXE$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 **class driver request packet** (CDRP), an 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 6–18 and are described in Table 6–6. Note that the fields of the figure run right to left.

Figure 6–18: Format of the AST Control Block

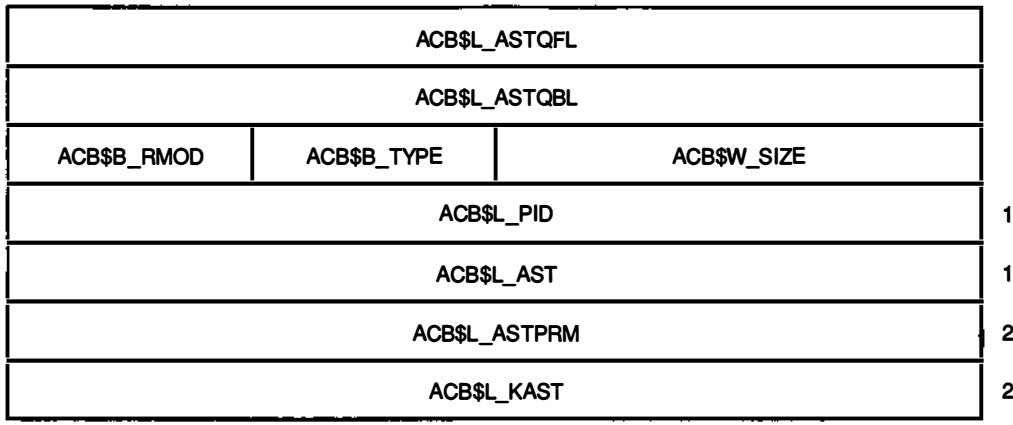


Table 6–6: Contents of the AST Control Block

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 through 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$L_DISPATCH.	
ACB$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. This routine is called to add a packet to the queue. 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$WAKACP in IOCIOPOST. This routine is called to start up the XQP. Like the code in EXE$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; the XQP can be entered and exited only with a special KAST.

```
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
```

6.5 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.

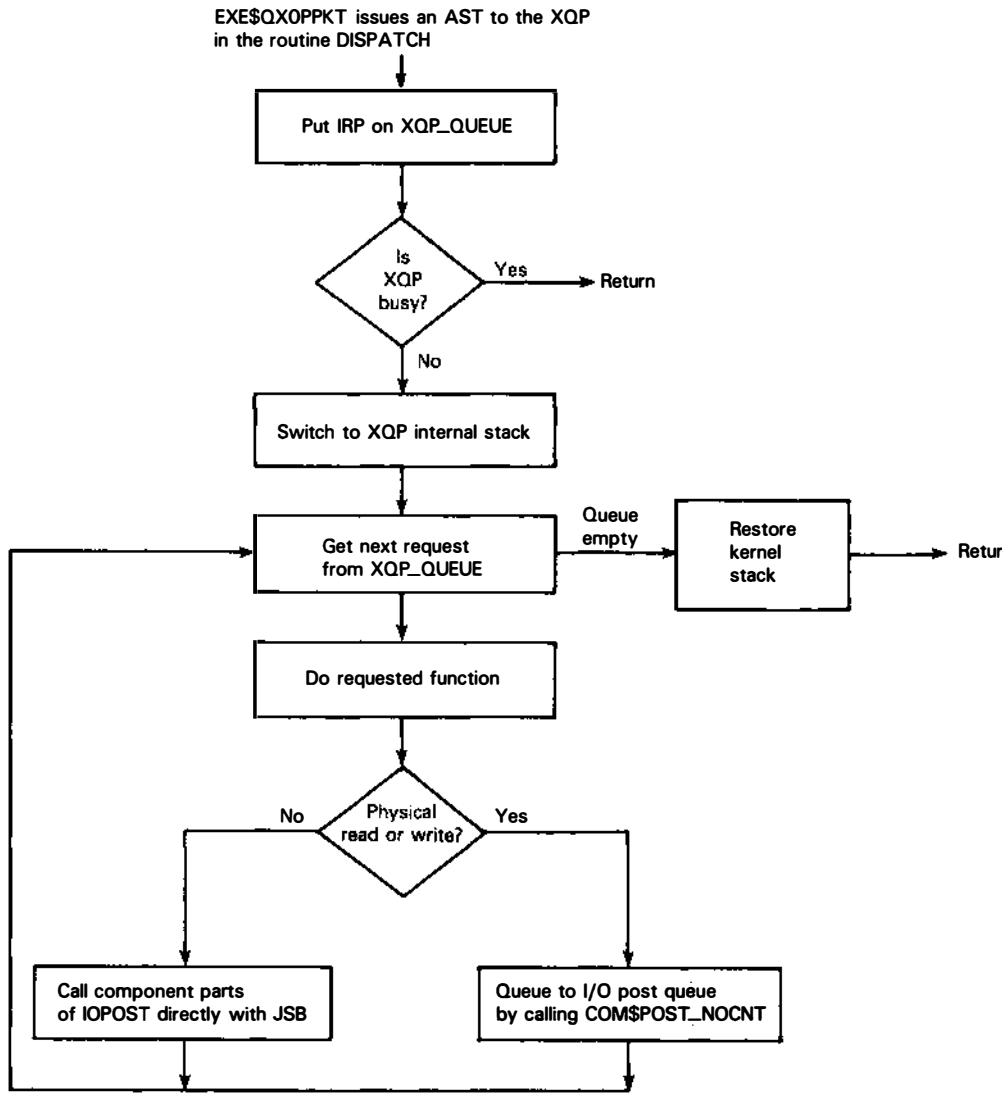
- DIRPOST via IOC$WAKACP

  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 6–19 illustrates the flow of code through the XQP after the AST has been sent to DISPATCH.

Figure 6–19: Overview of XQP Code Flow



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$GL_F11BXQP). All XQP routines assume that R10 points to the CONTEXT_START offset in the XQP storage area. The IRP is then queued onto a per-process queue in the impure area called XQP_QUEUE. F11B$Q_XQPQUEUE is another label for this queue.

The PCB$B_DPC cell is incremented to prevent process deletion while any file system request is being processed. This field must contain a 0 before the EXE$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 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.

6.5.1 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, Performance Monitoring Services (PMS) metering 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 value for PRIMARY_FCB is set if a window exists; a window does not exist for access, create, or mount functions.

If the I/O request is a normal file system request and not a window turn (that is, the IRP$V_COMPLX bit is set), the byte count for the window block descriptor (ABD$C_WINDOW) 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_BFR_CREDITS routine (see Section 5.2.2).

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 the 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 the IPL is lowered to 0.

If a blocking lock already exists, the 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 completed, PERFORM_AUDIT is called. See Section 6.6.3 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 6.7.

6.5.2 Processing in Secondary Context

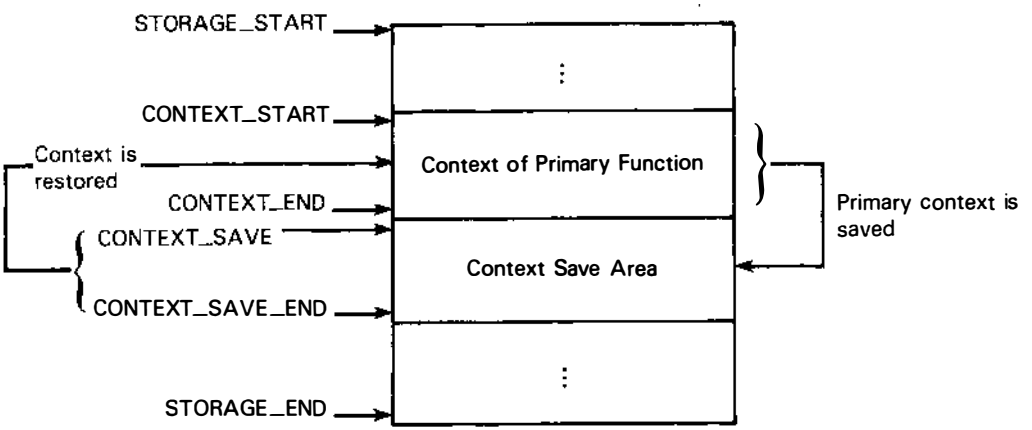
Some file system functions require what are called **secondary functions.** A secondary function is a normal file system function that is generated by, or on behalf of, another file system function, called a **primary function.** The primary function is not necessarily dependent on the results of the 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 6–20 shows how the primary context is copied from the primary function area to the context save area.

Figure 6–20: Saving and Restoring Primary Context in the XQP Impure Area



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 4 and 7). Also, any unrecorded blocks must be recorded before leaving secondary context (refer to Section 6.5.2).

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

- 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 5.4.8 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 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.

6.5.3 Switching Stacks

The XQP has an independent operating stack in the impure area that it uses when processing an XQP request. This stack allows the XQP to operate as an asynchronous kernel-mode thread. When the XQP has to wait for an I/O or lock request to complete, for example, it leaves its operating context on its private stack, switches back to the normal kernel stack, and returns to the caller. When the I/O or lock request completes, the XQP resumes operation on its internal stack.

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.

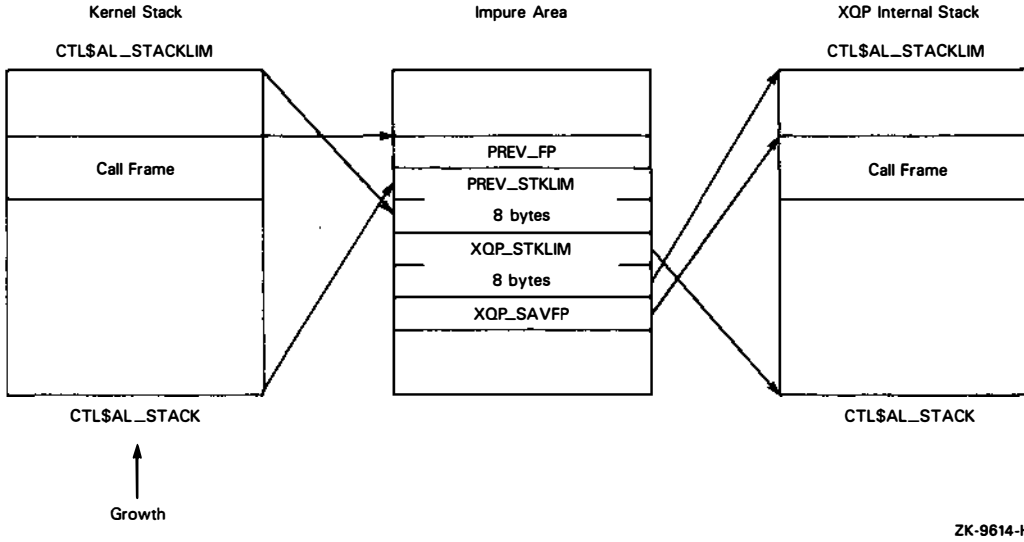
DISPATCH then sets the first longword of XQP_STKLIM, the base of the private XQP kernel stack, to be CTL$AL_STACK. It also sets the second longword of XQP_STKLIM to be CTL$AL_STACKLIM. XQP_STKLIM also becomes the new stack pointer, which initially points to the base of the private XQP kernel stack.

A list of the pointer updates that occur when the XQP switches from the normal kernel stack to its own private stack follows. 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 6–21 shows how the XQP switches from the normal kernel stack to its own internal stack. Note that the stack in this figure grows from bottom to top.

Figure 6–21: Switching from the Kernel Stack to the XQP Internal Stack



6.5.4 Stalling a Transaction

Because many copies of the XQP run on a system, 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.

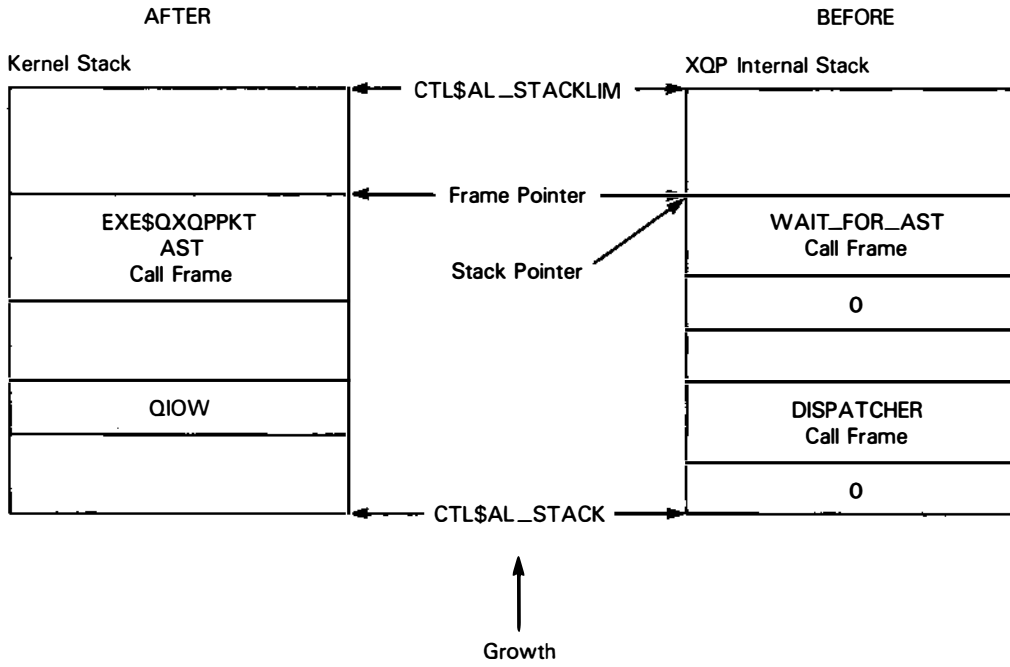
Performance Monitoring Services (PMS) metering is stopped for the duration of the stall.

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 6–22 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 6–22: Stalling a Transaction



ZK-9615-HC

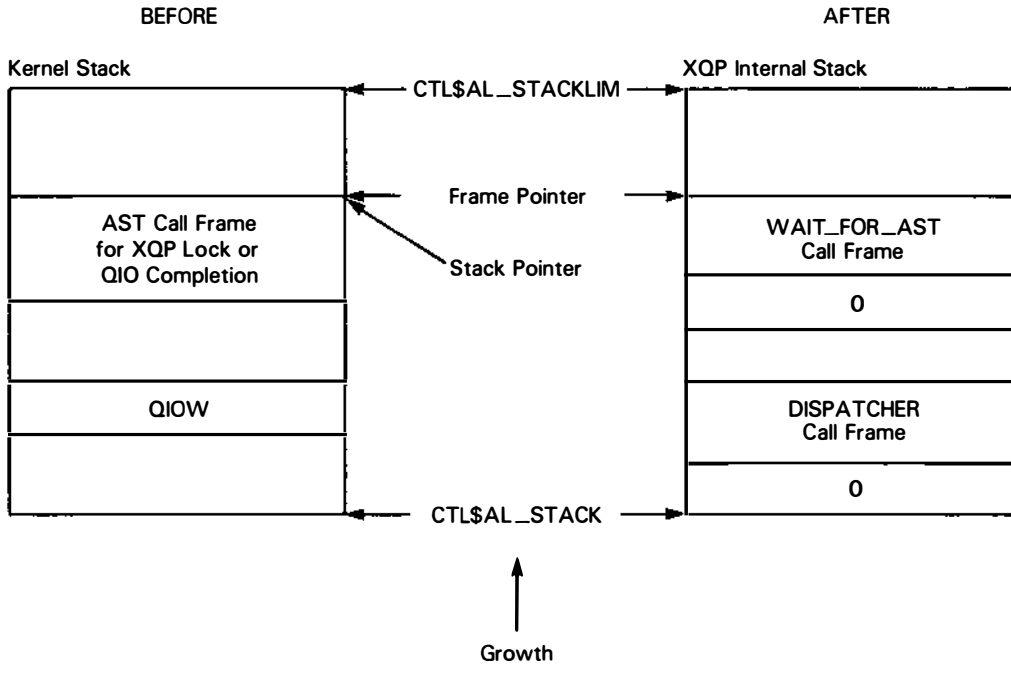
The QIO or ENQ request that was queued specifies the impure pointer (contained in R10) as the AST parameter and the routine CONTINUE_THREAD as the AST routine. CONTINUE_THREAD resets the kernel stack limits to the XQP private stack and restores the saved frame pointer. It then returns to resume execution 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 6–23 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 6–23: Unstalling a Transaction



6.6 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 do a 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 many times before the file system gives up. ERR_CLEANUP is also responsible for cleaning up secondary context.

6.6.1 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 responsible 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 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 clusterwide, if requested. The FCBs are purged unless they are currently accessed or directory index blocks are associated with them.

6.6.2 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.

6.6.3 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[1] 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.

[1] Unsupported.

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 not exist to be audited after MARK_DELETE operates.

6.7 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$L_MEDIA (which is actually a quadword).
- Decrements the transaction count for the VCB.
- Clears the name string descriptor length in the complex buffer packet to prevent the name from being written back for efficiency.
- Copies the local FIB back into the complex buffer packet. The FIB contains a variety of values which are returned to the user, such as the file ID, final file size, status flags, and so on.
- Sets IRP$L_BCNT to ABD$C_ATTRIB (if the function was not a read function[1]) to prevent the attribute list descriptors from being written back to the user's buffers. Attributes are only returned to the user on an IO$_ACCESS function. The IRP$V_FUNC bit is then set to cause IOPOST to copy the remaining descriptors back to the user's buffers.

IO_DONE finishes posting I/O completion for the request and then calls CHECK_DISMOUNT (see Section 6.7.3).

The I/O completion routines in IOCIOPOST are called in different ways, depending on whether the I/O being completed is a file function or a transfer function:

- Transfer functions are completed by posting a software interrupt to cause postprocessing to be executed on the primary CPU in a multiprocessor system. This guarantees that transfer operations complete in the proper order.

[1] The IRP$V_FUNC bit is clear.

- File functions, on the other hand, are completed by calling the component routines of IOCIOPOST directly. This saves the overhead of an AST and guarantees that posprocessing will be complete before IO_DONE calls CHECK_DISMOUNT.

6.7.1 Completing File Functions

Because the XQP performs 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 the BUFPOST routine for XQP functions (except window turns) requiring a complex buffer and buffered I/O.

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:

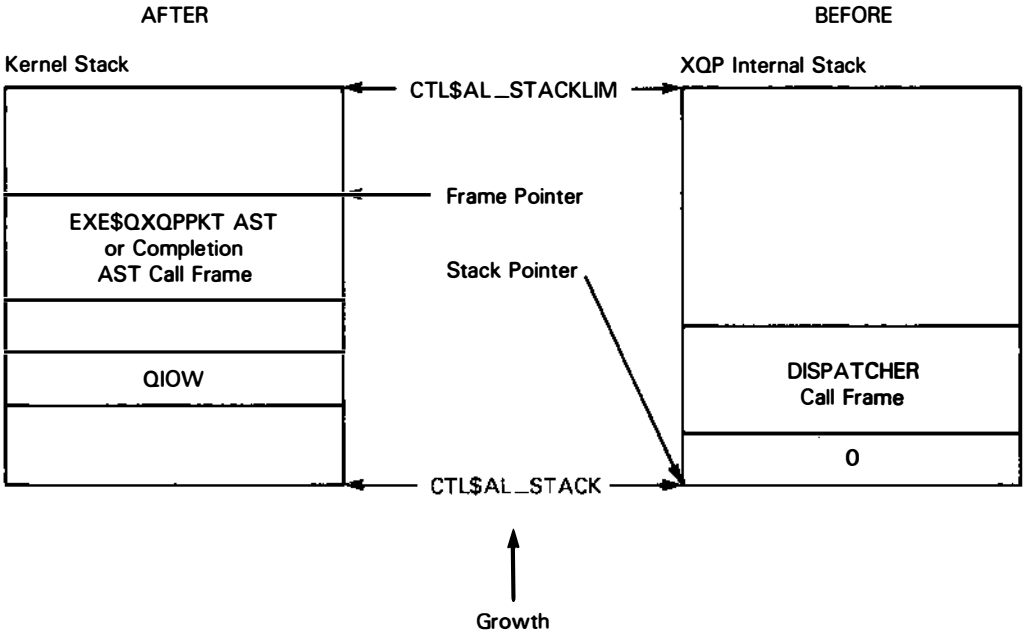
- Updates process header quotas (PHD$L_DIOCNT is incremented).
- Decrements the channel activity count in CCB$W_IOC, showing that there is no more I/O in progress.
- Sends a deaccess request to the XQP if the activity count is zero and CCB$L_DIRP indicates a pending deaccess function (CCB$L_DIRP contains a nonzero 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 and buffer packet.

Finally, the DISPATCHER routine calls FINISH_REQUEST. This routine sets the 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 the 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 inaccessible, and returns.

Figure 6–24 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 6–24: XQP Transaction Completion



6.7.2 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.

6.7.3 Checking for Dismount

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).

CHECK_DISMOUNT performs the following actions:

- Sets the UCB$V_DISMOUNT bit while the I/O database is locked to prevent other processes from starting I/O on the volume.
- Issues an IO$_UNLOAD/IO$_AVAILABLE function.
- Obtains the value block for the volume lock in protected write mode if the volume is mounted clusterwide.
- Clears the high bit of the UCB$W_DIRSEQ field to warn RMS of the volume dismount (for more information, refer to Section 8.6.6).
- Decrements the UCB$W_REFC field.

- Decrements the AQB$W_MNTCNT field and removes the AQB from the AQB list if the field goes to 0.
- Deallocates all FCBs, ACBs, and WCBs.
- Dequeues access locks by forcing FCB$W_REFCNT to 0.
- Dequeues the FID and extent cache locks, and deallocates the caches.
- Dequeues the quota cache lock, and deallocates the quota cache.
- Dequeues the volume lock (VCB$L_VOLLKID).
- Dequeues the shadow lock.
- Clears the RVT list entry for volume sets, and decrements the RVT$W_REFC field. If it is 0, 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.
- Dequeues the blocking lock for single volumes.
- Deallocates the VCB.
- Demotes the device lock, if any, either to concurrent read mode if the volume is not allocated or to exclusive mode if it is. The value block is cleared if this is the final dismount.
- Calls the routine IOC$DALLOC_DMT in the SYS module IOSUBPAGD to deallocate the device.
- Decrements the AQB reference count. If it goes to 0, the buffer cache is deallocated.

Chapter 7

Serialization of File System Activity

Serial adj. *Being or pertaining to just one damned thing after another.*
Stan Kelly-Bootle

Suhor's Law *A little ambiguity never hurt anyone.*
Charles Suhor

Outline

7.1 Introduction

When there is concurrent file system activity, the potential for overlap exists. The goal of turning concurrent file system activity into **serial** activity is to define a mechanism within the directory hierarchy and the lock structure so that some concurrencies can occur while others can be prohibited. The way the XQP coordinates, or **serializes**, this activity is by using the distributed lock manager.

7.2 Distributed Lock Manager

The XQP is mapped into the virtual address space of every process, and it executes operations for that process in its own process context. For the most part, XQP processes synchronize with each other using the same mechanism available to users in general: the distributed lock manager. Thus, each process is able to hold its own locks, queue its own locks, wait for its own resources, maintain its own data, and so on, in the normal process sense.

The file system uses a number of locks in a strict hierarchy to control and communicate dynamic information about files, caches, volumes, and usage. For most file system operations, the order in which locks are acquired, converted, manipulated, and released is strictly defined.

The file system extensively uses the distributed lock manager to synchronize file activity. If one process is performing some file system activity, a second process is prevented from performing the same activity until the first process is finished.

Locks can be held in a variety of modes. The available locking modes are mapped directly onto file access and sharing modes. The various access and sharing combinations are controlled by lock modes as shown in the following table.

Lock Mode	Meaning	Description
LCK$K_EXMODE	Exclusive	Grants read and write access to the resource and prevents the resource from being shared with any other readers or writers. This lock is the traditional "exclusive lock."
LCK$K_PWMODE	Protected write	Grants write access to the resource and allows the resource to be shared with concurrent readers. No other writers are allowed access to the resource. This lock is the traditional "update lock."

Lock Mode	Meaning	Description
LCK$K_PRMODE	Protected read	Grants read access to the resource and allows the resource to be shared with other readers. No writers are allowed access to the resource. This lock is the traditional "share lock."
LCK$K_CWMODE	Concurrent write	Grants write access to the resource and allows the resource to be shared with other writers. This lock mode is typically used to perform additional locking at a finer granularity, or to write in an "unprotected" fashion.
LCK$K_CRMODE	Concurrent read	Grants read access to the resource and allows the resource to be shared with other writers. This mode is generally used when additional locking is being performed at a finer granularity with sublocks, or to read data from a resource in an "unprotected" fashion (allowing simultaneous writes to the resource).
LCK$K_NLMODE	Null mode	Grants no access to the resource. The null mode is typically used as an indicator of interest in the resource, or as a placeholder for future lock conversions. Null mode is compatible with all other modes, so it ignores all other lock modes. Essentially, queuing a null-mode lock allows all other lock modes to be overridden.

7.2.1 Locking Conventions

The file system cannot deal with true concurrent activity, so locks are used as a means to serialize. A lock is not a form of permission; rather, it is a block or a preventive. Although in reality a lock is merely a portion of memory, it represents something larger and more significant—access to a resource. The process that holds the lock also holds the power to access (and to modify) the resource.

Locking before accessing a resource is a convention. It is not strictly necessary and can be overridden. However, locking conventions presume that if locks are not correctly obtained, the potential exists to corrupt data and to destroy the important data structures of the file system.

The following types of locking are used in the VMS kernel:

- **Spin locks**—Provide low-level protection against concurrent access to a resource by multiple CPUs. Spin locks are so named because the waiting CPU "spins" on the lock, repeatedly testing it until the lock holder frees it. Spin locks are held for relatively short periods of time and only at an elevated IPL. They are a generalization of, and replace, the old IPL synchronization design used before symmetric multiprocessing (SMP) was introduced in Version 5.0. Each spin lock corresponds to a particular IPL; the spin lock prevents concurrent access by another CPU, and the elevated IPL prevents delivery of conflicting interrupts on the same CPU.
- **Mutexes**—Provide low-level protection against concurrent access to a resource by multiple processes. To use a mutex, a process must be executing at IPL2 in kernel mode. When a process waits on a busy mutex, it goes into MWAIT state and is resumed by the scheduler when the mutex is made available. Mutexes are used for resources in paged memory and other resources that must be held too long to be appropriate for a spin lock.
- **Distributed lock manager locks**—Provide high-level protection against concurrent access to any resource by multiple processes. Lock manager locks are the only mechanism for synchronizing among the members of a VAXcluster system. They also provide a variety of powerful features such as multiple lock modes, value blocks, and blocking ASTs.
- **XQP-internal locks**—Used by the XQP to provide its own style of synchronization for the components of the buffer cache. Processes wishing to synchronize insert an AST control block on a queue; the first process on the queue is permitted to execute. When the executing process wishes to release the lock, it uses the next-queued ACB to wake up the next waiting process.

7.2.2 Distributed Lock Manager System-Owned Locks

A variation of the lock manager lock commonly used by the file system is the **system-owned lock**. The primary feature of a system-owned lock is that it is not owned by any particular process but by the system as a whole. A system-owned lock does not disappear when the process that created it terminates; it is removed only by explicitly dequeuing it. System-owned locks have two major uses:

- To synchronize resources whose lifetimes are independent of individual processes
- To combine the multiple locks of several processes into one lock

The following two system-owned locks are used to serialize file system activity:

- The volume allocation lock
- The serialization lock

These locks are maintained both as system-owned and as process locks under different circumstances.

7.2.2.1 Volume Allocation Lock

The **volume allocation lock** controls volume allocation and deallocation. It is the top-level lock for the volume, and it synchronizes critical operations on the volume. For example, the volume lock serializes operations on the index and storage bitmaps (that is, the allocation of free space and file IDs) and on their related caches (the FID cache and extent cache).

Because the volume allocation lock name is unique for any given volume or volume set, it is the logical choice as the parent lock for the file serialization lock (see Section 7.2.2.2 for information on the serialization lock). It can also be generated from any node in a VAXcluster from which a shared volume has been mounted.

This lock is the first on the volume to appear and the last to disappear. It exists longer than any other lock because all file activity must stop before the volume can be dismounted.

Along with the device name, the volume name is also used by the Mount Utility to enforce the requirement that two volumes with the same name cannot be mounted shareable at the same time in a VAXcluster.

Any number of volumes with the same volume label can be mounted privately. In that case, combining the system name and the address of the UCB for the device forms a name guaranteed to be unique throughout the VAXcluster.

The resource name used is the character string **F11B$v**, followed by one of three volume identifiers:

- The volume label, if the volume has been mounted shareable (that is, systemwide).
- A combination of the node name and the UCB address if mounted privately. When a volume is mounted with the /NOSHARE qualifier, the volume does not interact with other volumes on the system, so the UCB address is used as part of the volume lock name to guarantee both that the name does not duplicate a legitimate volume name and also that the name is unique.
- The volume set name, if this is a volume set.

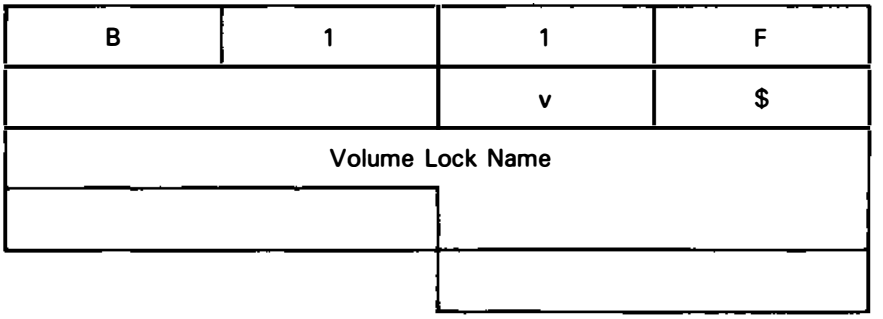
Using the volume label or volume set name for shareable mounts also enforces uniqueness of volume labels across the VAXcluster system.

The volume allocation lock may have the following forms:

```
F11B$v<volume ID>
F11B$v<system name><UCB address>
F11B$v<volume set ID>
```

Figure 7–1 shows the internal representation of a volume allocation lock.

Figure 7–1: Format of a Volume Allocation Lock



ZK-9618-HC

The 12-byte volume identifier part of this lock name is read from disk by the routine GET_VOLUME_LOCK_NAME when the volume is mounted. It is stored in the VCB field VCB$T_VOLCKNAM for later use by MOUNT and the XQP. (See Section 3.3.1.1 for more information on the volume control block.)

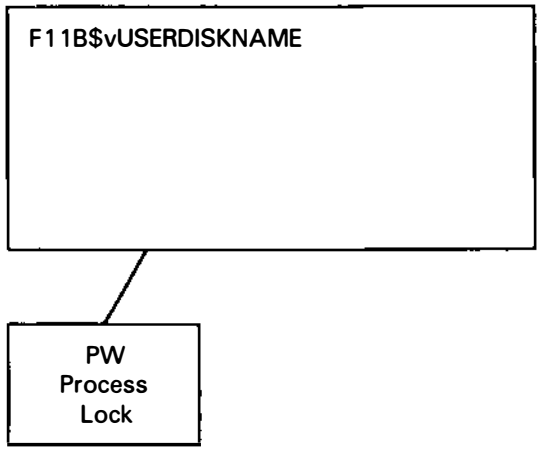
The volume identifier can be obtained with the DEVLOCKNAME item code of the $GETDVI system service. The item code actually returns a 16-byte field. One of the extra bytes is used to distinguish which volumes have been mounted shareable and which have been mounted privately to prevent possible duplicate names. Note that the volume lock name is stored in the VCB because the actual volume name can be changed with the SET VOLUME command after the volume has been mounted.

The file system avoids any interference between private and shared volumes by using the system-specific lock as a parent for privately mounted volumes. This approach also has the advantage of not requiring VAXcluster traffic to determine on which system the lock is mastered, which is a performance gain. The system lock ID is contained in the executive cell EXE$GL_SYSID_LOCK.

The volume lock ID is stored in the field VCB$L_VOLLKID.

The volume allocation lock is initially acquired in protected write mode by the MOUNT routine GET_VOLUME_LOCK (in CLUSTRMNT). Figure 7–2 shows the first stage in the life cycle of this lock.

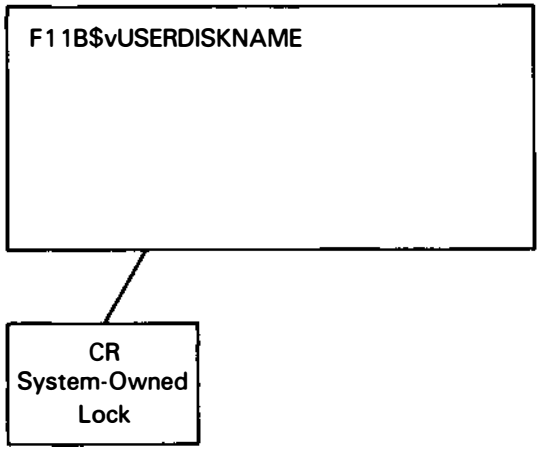
Figure 7–2: First Stage in the Volume Allocation Lock Life Cycle



ZK-9619-HC

It is converted to a system-owned lock in concurent read mode by the STORE_CONTEXT routine when MOUNT processing is complete. Figure 7–3 shows the second stage.

Figure 7–3: Second Stage in the Volume Allocation Lock Life Cycle

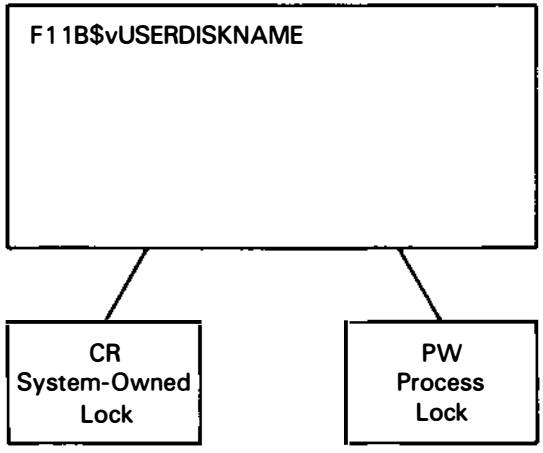


ZK-9620-HC

The lock remains granted in concurrent read mode until the volume is dismounted. It is dequeued by the CHECK_DISMOUNT routine when the transaction count on the volume becomes idle after the volume has been marked for dismount.

The volume lock remains permanently granted as long as the volume is mounted. It serves as the parent lock for all file number serialization locks. To synchronize volume operations, the volume allocation lock is taken as a process lock in protected write mode. Figure 7–4 shows the last stage.

Figure 7–4: Third Stage in the Volume Allocation Lock Life Cycle

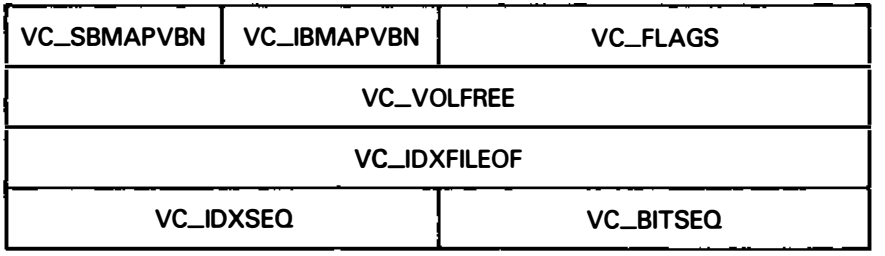


Because the volume lock or volume set lock is used as the parent lock for all synchronization locks, its lock ID also appears as the parent lock in BFRLs for blocks read on that volume or volume set.

In addition, the volume lock value block contains the current free space on the volume as well as some pointers to manage allocation. In effect, it acts as a rotating pointer in the storage bitmap to find free space in the bitmap more quickly.

Figure 7–5 shows the format of the volume allocation lock value block.

Figure 7–5: Format of the Volume Allocation Lock Value Block



ZK-9622-HC

Table 7–1 shows the contents of the volume allocation lock value block.

Table 7–1: Contents of the Volume Allocation Lock Value Block

Value Block Field Name	Description		
VC_FLAGS	Status flags. The following flag bits are defined within VC_FLAGS:		
	VC_NOTFIRST_MNT	Bit <0>	First mounted. This bit is clear if the volume has not been mounted elsewhere in the VAXcluster.
	VC_QUOTASEQ	Bits <1:15>	Sequence number for quota file data blocks. This field is incremented to invalidate the quota cache entry.
VC_IBMAPVBN	Current VBN in the index file bitmap. This field contains the virtual block number of the block at which to start the next file creation scan. It is taken from the value in VCB$B_IBMAPVBN.		
VC_SBMAPVBN	Current VBN in the storage bitmap. This field contains the virtual block number of the block at which to start the next allocation scan. It is taken from the value in VCB$B_SBMAPVBN.		
VC_VOLFREE	Number of free blocks on the volume. The value in this field is taken from VCB$L_FREE.		

(continued on next page)

Table 7–1 (Cont.): Contents of the Volume Allocation Lock Value Block

Value Block Field Name	Description
VC_IDXFILEOF	End-of-file VBN for the index file. This field is obsolete.
VC_BITSEQ	Sequence number for the storage bitmap. This field is incremented to invalidate all cached entries.
VC_IDXSEQ	Sequence number for the index file bitmap. It is incremented to invalidate all cached entries.

7.2.2.2 Serialization Lock

The **serialization lock** controls the serialization of file system processing on individual files. It is a sublock, and its parent lock is the volume allocation lock associated with a volume or volume set. It is held while the file system is changing the state of a particular file.

For example, the serialization lock is held to accomplish the following tasks:

- Extending a file
- Accessing a file
- Deleting a file
- Performing a directory operation

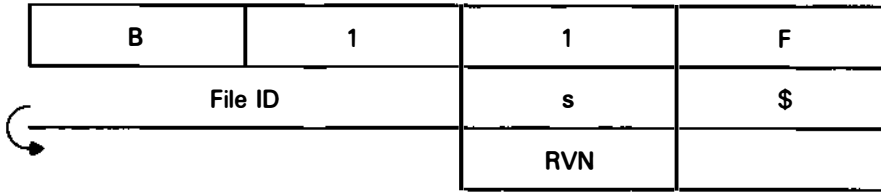
While this lock is actually held, it is held as a process lock to synchronize activity. Its purpose is to prevent two processes from making simultaneous modifications to the same file or directory.

Like the volume allocation lock name, the serialization lock name also uniquely identifies a file on a volume set or single volume. The resource name used for a serialization lock is the string **F11B$s**, followed by the 3-byte file number (a 2-byte file number and a 1-byte file number extension) plus a 1-byte relative volume number. It has the following form:

```
F11B$s<file number><relative volume number>
```

Figure 7–6 shows the internal representation of a file serialization lock.

Figure 7–6: Format of a Serialization Lock



ZK-9635-HC

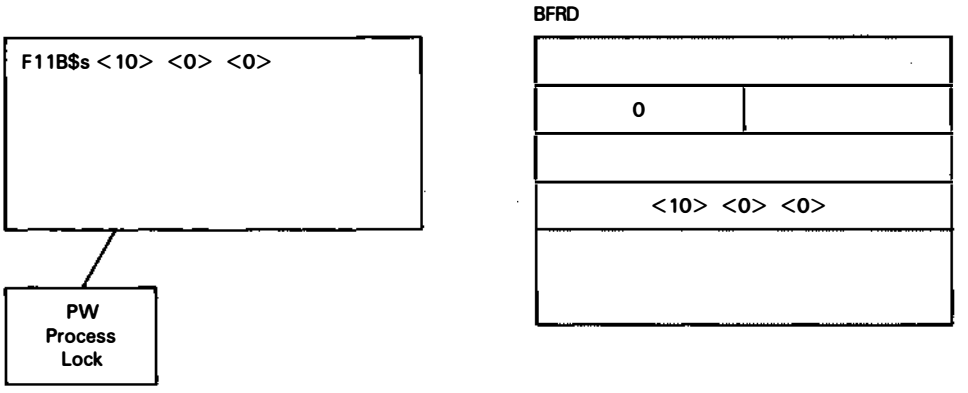
File serialization locks are taken out by the routine SERIAL_FILE. Either the volume lock or the volume set lock is specified as a parent lock depending on whether the file is on a single disk volume or a volume set.

The file serialization lock is taken out when the on-disk structures for a file (such as the file header) and the in-memory data structures (such as the FCB) are modified.

Like the volume lock, the serialization lock is always queued in protected write mode. Its completion AST is the XQP routine CONTINUE_THREAD, and its ASTPRM is the base register of the XQP impure area (R10). Processes must queue for ownership of this lock.

Figure 7–7 shows the protected write lock resulting from the first access of a header. After the lock has been granted, a buffer descriptor is allocated, and the header block is read in.

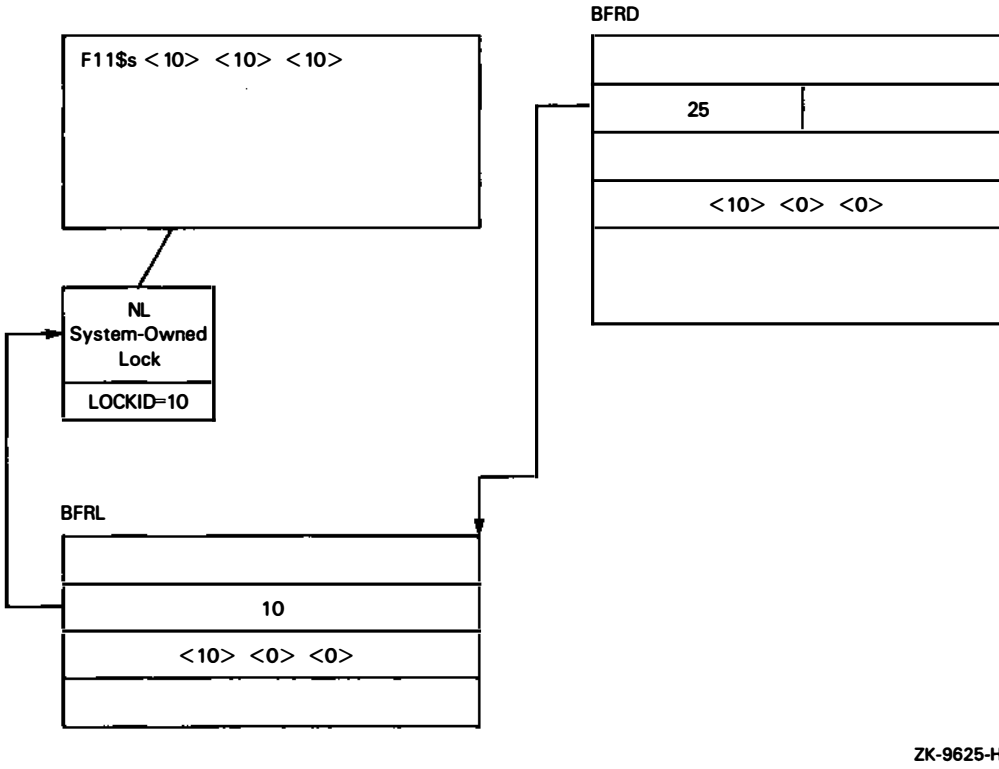
Figure 7–7: First Stage in the Serialization Lock Life Cycle



The serialization lock is released when the XQP is finished operating on the resource it represents. To coordinate with other nodes of the VAXcluster that may want to acquire or convert the lock, a BFRL structure is used to keep track of the system-owned lock.

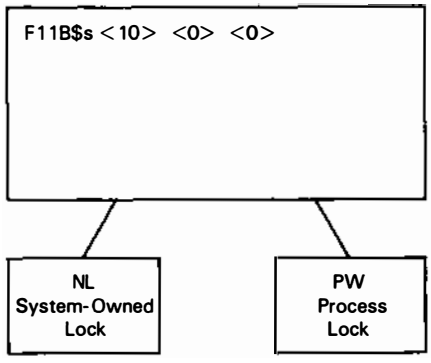
Figure 7–8 shows this stage of the lock life cycle. In this instance, the system-owned lock has a lock ID of 10, and the BFRL a buffer index of 25.

Figure 7–8: Second Stage in the Serialization Lock Life Cycle



When the system-owned lock is taken out, it remains until the header is flushed from the cache. Once that happens, a new owner can take out a protected write mode lock. Figure 7–9 shows this last stage in the lock life cycle.

Figure 7–9: Third Stage in the Serialization Lock Life Cycle



ZK-9626-HC

Figure 7–10 shows the format of the serialization lock value block.

Figure 7–10: Format of the Serialization Lock Value Block

FC_HDRSEQ
FC_DATASEQ
reserved
reserved

ZK-9627-HC

Table 7–2 shows the contents of the serialization lock value block.

Table 7–2: Contents of the Serialization Lock Value Block

Value Block Field Name	Description
FC_HDRSEQ	File header sequence number. This value is incremented if the primary or extent header is modified.
FC_DATASEQ	Directory file data sequence number. This value is incremented if any data blocks in the directory file containing the target file are modified. One number applies to all data blocks of the file. However, it does not apply to the directory file header. This field may also apply to bad block data.

The serialization lock is also used to synchronize the buffer cache. For more information, see Section 7.4.4.

7.3 Serializing Access to Files and Volumes

Every file has at least one file header, which is contained in the index file (INDEXF.SYS). Every file header has a unique set of numbers that identify it, and these numbers can be used to calculate the block in the index file that contains the corresponding file header. The file header for the index file is itself one of the blocks within the index file.

The file system extensively uses the distributed lock manager to synchronize file activity (such as updating the index file). Many routines and macros (such as SERIAL_FILE) exist to serialize activity, and they take as an argument the FID of the file to be locked.

During any file system function, any or all of the following disk volume structures may be referenced (in this order):

1. A **directory file**, including the directory header or the directory file data blocks. This is the file specified by the FIB$W_DID or directory ID, and it may be used to look up the target file in an access function or to make a directory entry in a create operation.
2. The target **file header** and any possible **extension headers**. Serialization locks for the directory and target files are handled by the SERIAL_FILE routine, which performs the following actions:
   - Takes the file ID as input and extracts the file number and relative volume number to construct the F11B$s lock resource name.

- Takes out the serialization lock.
- Returns an index into the vector LB_LOCKID, which keeps track of the lock ID of the lock granted. This index is stored in the following cells:
  - DIR_LCKINDX—For the directory serialization lock
  - PRIM_LCKINDX—For the target or primary file

In order to minimize both the number of locking operations and the number of locks required to perform a given operation, access to extension headers is not serialized with a separate lock but rather with a serialization lock on the primary header. In normal operations, this approach works because the primary header is always taken out first, and the link is followed from header to header.

Serialization locks on the directory and primary file are normally held until the completion of the entire operation. All locks are released in the routine UNLOCK_XQP, which is called from the DISPATCHER routine after the operation completes.

3. The **storage** and **index file bitmaps**. If caching is not enabled, the storage bitmap is used during the following operations:
   - When free storage is mapped to a file being extended or created
   - When free storage is returned from a file being truncated or deleted

   Likewise, the index file bitmap is scanned during the following operations:
   - When a new file or extension header is created
   - When a file header is deleted

   However, if caching is enabled, the extent cache and the file number caches are searched instead of their corresponding on-disk structures.

   Operations that involve the storage or index file bitmaps are serialized with an F11B$v lock. This lock is taken out by the ALLOCATION_LOCK routine, which is very similar to the SERIAL_FILE routine. The lock ID of this allocation lock is always stored in element 0 of the LB_LOCKID vector.

   Manipulating the headers of the storage or bitmap files requires taking out the serialization lock on the file as well as the allocation lock.

4. The **quota file**, which reflects allowed and current disk quota usages (if enabled).

7.4 Serializing Access to Shared Data Structures

The file system uses multiple synchronization mechanisms to serialize access to data structures that may be shared by multiple processes. For example, access to the following data structures is serialized:

- File control block
- Volume control block
- FID, extent, and quota caches
- Buffer cache

7.4.1 Serializing the File Control Block

The file control block is a volatile data structure that is shared by all the accessors of a particular file. Access to a file is controlled by two mechanisms:

- Taking out the serialization lock on the file and therefore the FCB
- Raising the IPL (spin lock)

7.4.1.1 Using the Serialization Lock to Serialize Access

The serialization lock synchronizes activity on a file after the FCB corresponding to the file has been found or created. (Access to buffers associated with a given file is also serialized by the F11B$s locks; however, this mechanism is not the same as that used to locate the buffers in the cache.)

The volume lock is the parent lock of the file serialization lock for efficiency and to save the overhead of moving locks around the cluster. Also, creating a sublock uses fewer resources than creating a top-level lock.

When a file is opened for the first time, an FCB is created. It exists from the time the first accessor in a VAXcluster opens the file until the time the last accessor in the cluster closes the file.

A file is represented by one FCB per node. If the FCB on one node is updated, all the other FCBs corresponding to that file must also be marked invalid. To signal that the information in the other FCBs is stale and must be invalidated, a blocking AST is fired. For further accesses, the file header must be read from disk and the FCB chain rebuilt. For these reasons, the FCB is a convenient data structure for those files that change most rapidly.

For example, the FCB is a practical structure for a situation in which one process on one node is reading and another process on another node is writing. In this case, the FCB does not have to be rebuilt often because the file is represented by a window control block (unless deferred truncation has been enabled). Therefore, the read operation can continue without reading from disk, and the write

operation can continue modifying the file (thus invalidating the other FCBs) and updating EOF markers.

However, if two processes try to write to the file simultaneously, there is contention for the disk. In this case, the write operations must be serialized by a lock instead of by the FCB.

A consistency bugcheck within the XQP is not necessary to guarantee that an appropriate serialization lock is held when an FCB is referenced.

7.4.1.2 Synchronizing Access to FCBs and WCBs

The majority of the FCB contents are used only by the file system and so are synchronized by taking the file's serialization lock before referencing or modifying the FCB.

However, the FCB list, rooted in the VCB, must be separately synchronized to prevent, for example, a process that is searching the FCB list from seeing an inconsistent list because another process has added or inserted an FCB. To accomplish this, the process must take the FILSYS spin lock while searching the FCB list or inserting or removing FCBs.

Other fields in the FCB such as file size and access counts are referenced but never modified outside the XQP. These fields are not synchronized but are updated conservatively such that outsiders see only safe values (for example, a file size that is never larger than the file is at that instant).

The WCB is used and modified by the XQP, and is scanned by IOC$MAPVBLK. It is also synchronized with the FILSYS spin lock.

7.4.2 Serializing the Volume Control Block

Another data structure to which access must be serialized is the volume control block, a static data structure. A VCB is created when a device is mounted, and it survives for the life of the volume. It is customized for each node at the time it is created, but once it has been built, much of the information stays the same (except for the volume label). Important information such as the UCB address does not change. The information in the VCB that does change, such as the volume free space count (VCB$L_FREE), is synchronized with the volume lock.

7.4.3 Serializing the File Number and Extent Caches

The file extent and file number caches (pointed to by VCB$L_VCA) are serialized by the F11B$v allocation lock. That is, access to those shared structures by multiple processes is controlled by the volume allocation lock.

In a VAXcluster, every node has an extent cache and a header cache. These special caches exist so a single node can get blocks without disrupting activity on the rest of the VAXcluster. When the cache is created, bits from the bitmap are divided among the local caches with no overlap. Local nodes can read and write their particular part of the cache without interfering with the rest of the bitmap. The convention is that when disk space is reserved to the cache, the corresponding bitmap bits are cleared (indicating that those blocks are not available for allocation). When the cache is flushed back to disk, then the bits truly reflect the disk blocks that are available or allocated.

Additional locks are used to cause the special caches to be flushed under appropriate circumstances, such as when access to related files would interfere with the cache or when the resources are needed elsewhere in the cluster. Such cache-flush-and-fill operations are also done under the volume lock to synchronize access to both the cache and the related resource. Refer to Section 8.3.3 for more information on the cache flush lock.

7.4.4 Serializing the Buffer Cache

Changing the state of the buffer cache descriptors in the overhead area must be done atomically so that any process needing to use the cache always sees a consistent picture. Searching or manipulating the cache must therefore be serialized.

Serial access to the buffer cache only needs to be enforced for the processes on a single node, however, not across an entire cluster. Serial access to the buffer cache is achieved without locking (the INSQUE instruction is used). Because the lock manager is not involved, cache serialization can be performed very quickly.

Serialization is controlled by the AQB$L_ACPQFL queue, and the first entry (an IRP) in the queue has control of the cache. An ACB in the class driver extension to the IRP (CDRP) is set up to point to the XQP routine CONTINUE_THREAD.

The two routines that acquire and release the cache interlock are SERIAL_CACHE and RELEASE_CACHE, which are called by the other routines in the RDBLOK module. SERIAL_CACHE performs the following actions:

- Queues the IRP at the tail of the AQB$L_ACPQFL queue.
- If the queue is empty, returns so that its caller can proceed.
- Calls the WAIT_FOR_AST routine if another entry is on the queue. This routine returns only after obtaining ownership of the cache.

The RELEASE_CACHE routine performs the following actions:

- Removes the IRP from the head of the queue when the transaction is complete.

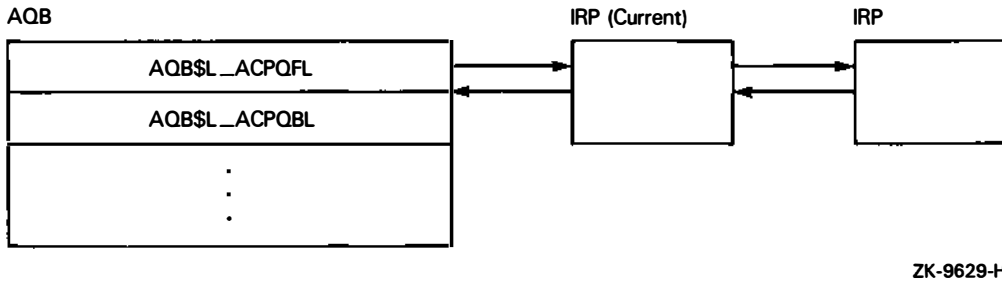
- If another IRP is found on the queue, uses the CDRP portion of the IRP to queue an AST to the waiting process.

The CDRP area of the IRP is again used as an ACB, just as it was used to start the AST thread in the first place. When it is inserted in the queue, SCH$QAST (queue restart of first IRP in queue) is called.

The cache serialization interlock is held only while the cache is searched or the state of the buffer descriptors is changed. It is never held when the XQP must stall for I/O to finish or for any other reason.

Figure 7–11 shows how access to the block cache is serialized.

Figure 7–11: Serializing Access to the Block Cache



7.5 Deadlock Considerations

The file system is designed to be free of deadlocks. By assuming a hierarchical directory and file structure, taking out locks in the order prescribed in Section 7.3 results in a deadlock-free system.

Certain operations, such as creating a new file, must access files in a different order. Specifically, the allocation lock must be taken out first to determine the file number of the primary file to be created, and then the directory entry can be made.

In this case, deadlock is avoided by releasing the allocation lock prior to acquiring the serialization lock on the new file header. In general, the allocation lock is always released before a new file number serialization lock is acquired. The ALLOCATION_UNLOCK routine performs this function.

The FID_TO_SPEC routine is an even more pathological example of needing locks in the wrong order because it walks the directory hierarchy backwards.

The serialization lock can be held on a newly created file before the directory serialization lock is taken out (even though it violates the ordering rule). The reasoning is that if this is a new file, no other users should be able to find it in the directory.

7.6 File System Internal Serialization Checks

To enforce the requirement that an appropriate serialization lock be held when a header is read from disk, there is another vector called **LB_BASIS** that uses the lock index returned from the SERIAL_FILE routine. LB_BASIS contains the **lock basis**, or file number plus the RVN field, of a given serialization lock.

The last lock index returned by SERIAL_FILE is contained in the XQP impure area cell CURR_LCKINDX. The READ_HEADER routine uses these bits of information, in addition to looking at the header just returned from the READ_BLOCK routine, to determine whether the correct serialization lock is held for the header just requested. This behavior is enforced because the READ_HEADER routine is used by all XQP code to read a header.

For example, if a user specifies an extension header by file ID and attempts a delete function on the extension header, the following actions occur:

- The given file ID is serialized by the MARK_DELETE routine.
- The CURR_LCKINDX and LB_BASIS fields are filled in by SERIAL_FILE.
- The header is obtained by READ_HEADER.
- An extension header is detected by READ_HEADER based on the FH2$W_SEG_NUM field in the header. Furthermore, the routine finds that the lock basis for the serialization lock does not match the primary header for that file (determined from the FH2$W_BK_FIDNUM and FH2$B_BK_FIDNMX fields).
- A status of SS$_NOSUCHFILE is returned as READ_HEADER exits, thereby making direct access to extension headers impossible.

There are exceptions to the rule of not allowing access by extension header, however. The Backup and Dump Utilities perform access functions explicitly on extension headers to get the extension header with a read attributes list that returns the complete file header. This is because both utilities need to have all the extension headers of a given file.

The ACCESS routine calls the READ_HEADER routine to handle this case. To perform an access function on an extension header, the following actions are performed:

- The ACCESS routine first takes out the serialization lock on the given file ID as if it were a primary header. This is allowed because the file system cannot tell whether the header is a primary or an extension header at this point.

- ACCESS calls READ_HEADER with an extra argument—an optional output from READ_HEADER. The extra argument tells READ_HEADER not simply to return a status of SS$_NOSUCHFILE if it encounters a lock basis mismatch, but rather to return the real lock basis for that extension header, derived from the primary header back link field.
- The incorrect serialization lock is released.
- The correct one based on the information from READ_HEADER is taken out.
- The whole operation is retried, including the call to READ_HEADER, because until the correct serialization lock has been taken out, the header could be changed at any time by another process.

7.7 File System Lock Indexes

The file system uses the following fields in the XQP impure area to keep track of three lock indexes:

Impure Area Field Name	Description
CURR_LCKINDX	Current file header lock index. This field is set by SERIAL_FILE to record the last index it returned. If RELEASE_SERIAL_LOCK is called with a lock index equal to CURR_LCKINDX, CURR_LCKINDX is zeroed. However, zero (the allocation lock) is not a valid value for these lock index variables.
PRIM_LCKINDX	Primary file lock basis index. This field is the index corresponding to the lock on the primary file header. Normally, PRIM_LCKINDX has the same value as CURR_LCKINDX. PRIM_LCKINDX is normally not itself referenced although ERR_CLEANUP forces CURR_LCKINDX to equal PRIM_LCKINDX. However, PRIM_LCKINDX does not always have the same value as CURR_LCKINDX. The lock basis corresponding to PRIM_LCKINDX can be wrong if an attempt is made to access an extension file header directly. PRIM_LCKINDX must be corrected (that is, the correct lock basis and lock must be obtained) in ACCESS.
DIR_LCKINDX	Directory lock basis index. This field records the index into the lock arrays (LB_BASIS and LB_LOCKID) for the parent directory of the operation. DIR_LCKINDX is not in the context area saved for secondary operations.

The following table shows some basic file system activities, which lock index is used to perform the activity, and the routine that sets the lock index:

Activity	Lock Index	Routine
Reading random file headers and blocks	CURR_LCKINDX	Used by READ_BLOCK. This lock is not normally released until request cleanup time.
Checking for the correct lock basis	CURR_LCKINDX	Used by READ_HEADER. This lock is not normally released until request cleanup time.
Removing blocks from a file into the BADBLOCK file	CURR_LCKINDX	Used by DEALLOCATE_BAD. If this lock is taken out in secondary context or on the index file in primary context (moving the EOF marker), it must be released separately. An explicit write of modified buffers is performed, and the lock is explicitly released (clearing PRIM_LCKINDX also).
Marking a file for deletion	CURR_LCKINDX	Used by MARK_DELETE. If this lock is taken out in secondary context or on the index file in primary context (moving the EOF marker), it must be released separately. An explicit write of modified buffers is performed, and the lock is explicitly released (clearing PRIM_LCKINDX also).
Serializing on the quota file (to rebuild stale FCBs)	CURR_LCKINDX	Used by SEARCH_QUOTA. The CURR_LCKINDX value must be saved during the rebuild operation because it refers to the quota file.
Performing quota file operations	CURR_LCKINDX	Used by QUOTA_FILE_OP. The quota file serialization lock (as well as the allocation lock) is held.
Advancing the index file EOF	CURR_LCKINDX	Refers to the index file serialization itself during the header write.
Remapping the index file	CURR_LCKINDX	Refers to the index file serialization lock.

Activity	Lock Index	Routine
Copying attributes from one file to another	PRIM_LCKINDX	Used by PROPAGATE_ATTR (in CREATE). In this routine, executed in secondary context, PRIM_LCKINDX points to the file from which attributes are copied.
		CURR_LCKINDX is saved across the OPEN_FILE call and points to the target file in case its headers must be reread (when buffers in which to write attributes are sought).
Creating a file	PRIM_LCKINDX	Used by CREATE. A serialization lock can be held from a previous access attempt (if this was a create-if access), so any PRIM_LCKINDX lock is released. (ACCESS, like most routines, does not clean up after itself.)
Purging the buffers for the extension headers	CURR_LCKINDX	Used by DELETE_FILE. A serialization lock is built on the extension header file ID as a basis for purging the buffers. The value of CURR_LCKINDX must be saved.
Accessing a directory	CURR_LCKINDX DIR_LCKINDX	Used by DIR_ACCESS. CURR_LCKINDX is saved while DIR_LCKINDX is being established (in a call to SERIAL_FILE).
Translating a file ID to a file specification via back links	CURR_LCKINDX PRIM_LCKINDX	Used by FID_TO_SPEC. This routine releases the PRIM_LCKINDX lock to avoid synchronization deadlocks when processes traverse the directory hierarchy.
		The reference count is incremented on the FCB, though, to maintain its existence. CURR_LCKINDX refers to the various directories in the back link chain. PRIM_LCKINDX is redetermined when control is returned to the file after the search.
Reading and writing virtual blocks	CURR_LCKINDX	Used by READ_WRITEVB. A serialization lock (CURR_LCKINDX) is obtained on a file ID when it is determined that a process is trying to directly write a file header.

Activity	Lock Index	Routine
Extending or compressing a directory	CURR_LCKINDX DIR_LCKINDX	Used by SHUFFLE_DIR. This function resets (in secondary context) CURR_LCKINDX to DIR_LCKINDX so that READ_BLOCK works.

7.8 Ambiguity Queue

It is possible to serialize on the wrong lock basis if an attempt is made to access an extension file header directly. If that same extension header is concurrently accessed by another process as an extension header using the correct lock basis, it is possible for one of those processes to locate the buffer in the cache that is in use by the other process, a condition known as **ambiguity**.

Normal file number serialization usually prohibits this behavior, and except for the specific case of file headers, it would cause an XQPERR bugcheck. However, in this case, the process puts itself on the **ambiguity queue** and goes to sleep.

Two situations in which ambiguity can occur is during BACKUP and DUMP processing. Both these utilities process extension headers directly, which is normally not allowed. They take out a lock on the primary header (which constitutes the lock basis); this lock also covers the extension header chain.

When the chain is locked, it can change unexpectedly even while it is being traversed. For example, an extension header could even switch files. So to handle this problem, the ambiguity queue is used.

The queue header F11BC$L_AMBIGQFL points to the ambiguity queue. When the XQP detects ambiguity, the RESOLVE_AMBIGUITY routine queues the current IRP onto the ambiguity queue, and the process goes to sleep. When the next operation completes, the RETURN_CREDITS routine awakens the process to try again. This cycle can happen any number of times until the ambiguity is resolved.

FIND_BUFFER detects the ambiguity case (when it finds a buffer in use in some other process). WRONG_LOCKBASIS and RETURN_CREDITS check the ambiguity queue for processes to waken.

Chapter 8

File System Operation in a VAXcluster Environment

That's our system, Nickleby; what do you think of it?
Charles Dickens

There ain't nothing more to write about, and I am rotten glad of it, because if I'd 'a' knowed what a trouble it was to make a book I wouldn't 'a' tackled it, and ain't a' going to no more.
Mark Twain

Outline

8.1 Introduction

Originally, a single-user system (batch with orthogonal file access) had no write-sharing problems. A solution for a single system with multiple users (that is, a timesharing system) was to allow only exclusive access for reading or writing. Eventually, users wanted time sharing with file sharing, and so their needs outgrew the standard multiple file access methodologies and various data organizations. Most of these techniques involved exclusive write access to a file but shared read access.

VAXcluster systems introduced new dimensions to these problems because multiple processes (on the same or different nodes) needed shared write and read access within the cluster. There were two major problems in a VAXcluster environment:

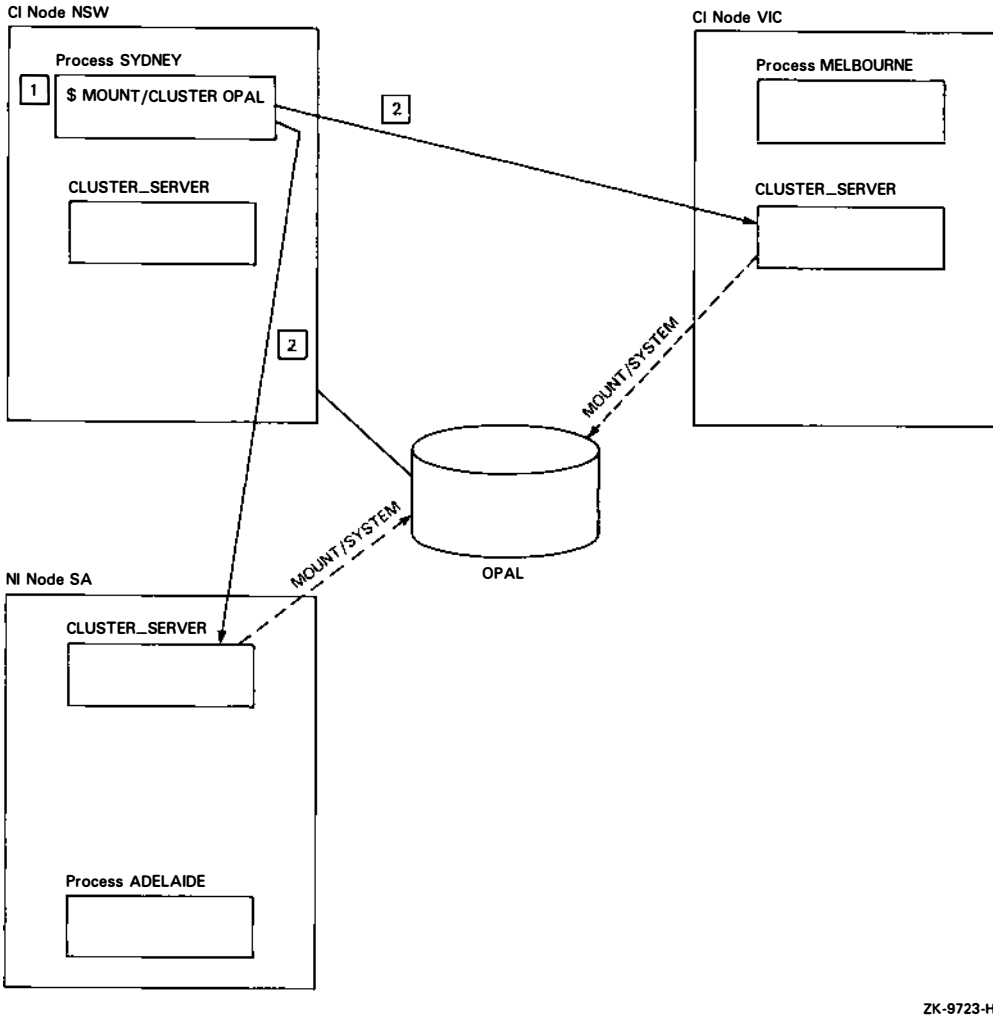
- Interprocess communication on the same node
- Interprocess communication among VAXcluster members

When the problem of distinguishing between processes for synchronization in the cluster was solved, the same synchronization problem per node was also solved. Both of these problems were solved by putting the XQP in process space and by extensive use of the distributed lock manager.

8.2 Mounting a Disk Clusterwide

Figure 8–1 shows how some of the VAXcluster components operate to mount a disk clusterwide.

Figure 8–1: Mounting a Disk Clusterwide



The file system performs the following steps when mounting a disk clusterwide:

1. OPAL is a disk on node NSW. Process SYDNEY on node NSW issues the command MOUNT/CLUSTER OPAL. This request is queued through SCS communication channels to all other nodes in the VAXcluster.

2. The cluster server process on nodes S_AUST and VIC respond by mounting volume OPAL on their nodes. The cluster server process on node NSW is not notified because the request came from that node.

8.3 Locking in a VAXcluster

In a VAXcluster, locks are used to control access. There are essentially two kinds of locks: system-owned and process-owned. A process-owned lock has the property that it is dequeued when the process vanishes. A system-owned lock, on the other hand, is not directly associated with any process. As a result, no process-specific functions can be communicated by means of the lock. Blocking ASTs are not delivered to a process; rather, they execute as interrupt-level JSB routines in kernel mode.

System-owned locks are an unsupported interface and Digital recommends that they never be used in any application or system environment. They are defined, designed, and maintained strictly for the internal use of VMS.

There are six system-owned locks, and the following five locks are covered in this chapter:

- Volume allocation lock
- Arbitration lock
- Blocking lock
- Cache flush lock
- Quota cache lock

The serialization lock is covered in Chapter 7.

Each of the special locks has a separate purpose. Generally, they must be acquired and released in hierarchical order. In addition, some are related in function. For example, the volume allocation lock and the quota cache lock are both affected when a file is extended. However, affecting what is protected by the quota cache lock does not necessarily mean that the volume allocation lock will be affected, and vice versa (quotas are not always enabled).

The various access and sharing combinations map into the distributed lock manager lock modes as follows:

Lock Mode	Meaning	File System Interpretation
LCK$K_EXMODE	Exclusive access	Read/write, disallow read/write
LCK$K_PWMODE	Protected write	Read/write, disallow write
LCK$K_PRMODE	Protected read	Read, disallow write
LCK$K_CWMODE	Concurrent write	Read/write, allow read/write
LCK$K_CRMODE	Concurrent read	Read, allow read/write
LCK$K_NLMODE	Null access	Ignore all

8.3.1 Volume Allocation Lock

The **volume allocation lock** is the top-level lock of the hierarchy, and it controls access to a volume. Top-level locks are mastered by the first node to acquire them, which contributes to locality of lock use. Like a number of the other locks, it serves several different purposes:

- Acts as the parent lock for most of the other locks on the volume.
- Indicates that a volume is mounted. For example, the Mount Utility uses the presence of the volume lock to prevent duplicate volumes from being mounted in a VAXcluster system.
- Passes context back and forth between VAXcluster nodes.
- Synchronizes operations that need to be synchronized on a volumewide basis. For example, the allocation of free space and file IDs is synchronized under the volume lock.

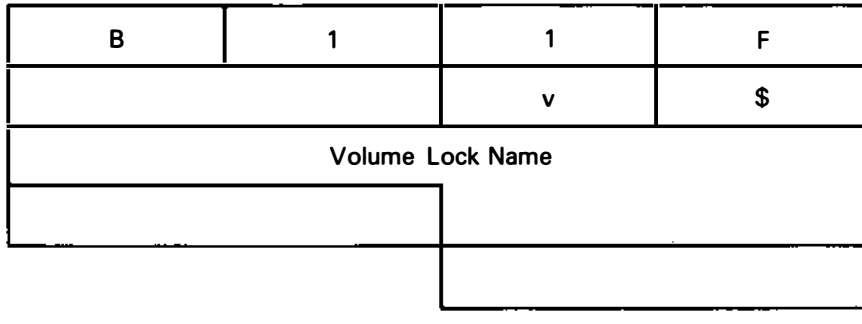
  In addition, the volume lock controls pointers to manage allocation. It serves, in effect, as a rotating pointer in the storage bitmap that the file system uses to find free space in the storage bitmap more quickly than could be done with I/O to and from the disk.

The volume allocation lock has the following form:

```
F11B$v<volume ID>
```

Figure 8–2 shows the in-memory representation of the allocation lock.

Figure 8–2: Format of a Volume Allocation Lock



ZK-9630-HC

The volume lock name is normally derived from the volume name of a volume that has been mounted systemwide. However, when a volume has been mounted with the /NOSHARE qualifier (which means that the volume cannot be mounted on other nodes in the cluster), the UCB address is used to ensure that the lock name does not conflict with another volume name. The UCB address also guarantees that the name is unique.

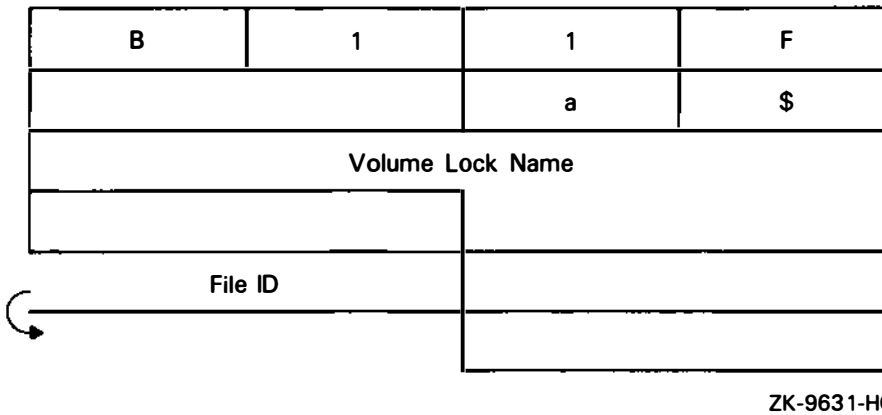
8.3.2 Arbitration Lock

The **arbitration lock** or **access lock** is the system-owned lock used to manage file access. It relates to a file in the same way the volume allocation lock relates to a volume. The access lock is a root lock, and it has the following form:

```
F11B$a<volume ID><file number>
```

Figure 8–3 shows the in-memory representation of the arbitration lock.

Figure 8–3: Format of an Arbitration Lock



ZK-9631-HC

The volume lock name is present because the arbitration lock is a top-level lock, not a sublock. The volume lock name or <volume ID> follows the rules for F11B$v locks, and the file number is the 3-byte file number (a 2-byte file number and a 1-byte file number extension).

The arbitration lock is mastered by the first node to acquire it, which contributes to locality of lock use. In general, for a file that is opened and not shared across the cluster, the arbitration lock is held and mastered only on the node on which the file is opened. This logic results in savings both in space and in lock-processing overhead. The value block of the arbitration lock is used to manage file truncation, and it is held in the mode in which the file was accessed.

The arbitration lock serves an alternate purpose. While a file is accessed, a blocking AST is enabled on the arbitration lock. This blocking AST is used to invalidate stale file control blocks clusterwide.

This technique may be used if, for example, a file is opened for shared write across the cluster and a process on one of the nodes extends the file, modifies the access control list, or otherwise changes one of the file attributes. File attributes are contained in the file control block, which, in effect, acts as a cache for the file header. The FCB is updated on the node on which the change is made, but it is not updated on the other nodes. These file control blocks then become stale on the other nodes, and must be updated from disk.

The file system corrects this situation by taking the following actions:

- The file system raises the arbitration lock to an incompatible lock mode.
- This enqueue operation causes the blocking AST to fire on the other nodes. Immediately after this request is enqueued, it is dequeued; the only purpose of the enqueuing operation is to cause the AST to fire.
- The blocking AST marks the file control block as stale.
- The file system will rebuild the FCB the next time it is referenced.

For more information on access arbitration, see Section 8.4.

8.3.3 Cache Flush Lock

The **cache flush lock** is used to force the following caches (under different circumstances) to write their contents back to disk:

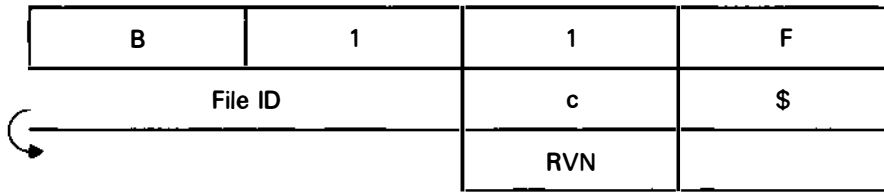
- The file ID cache
- The extent (bitmap) cache
- The quota cache

The cache flush lock has the following form:

```
F11B$c<file ID><relative volume number>
```

Figure 8–4 shows the in-memory representation of the cache flush lock.

Figure 8–4: Format of a Cache Flush Lock



ZK-9632-HC

The lock name is constructed from the **F11B$c** prefix, followed by a 3-byte file number and a 1-byte relative volume number. The FID that is used depends on the cache. Table 8–1 shows the three special caches, the FID used, and the file from which the FID was taken.

Table 8–1: Determining the FID in the Cache Flush Lock Name

Special Cache	File Number	Source
FID cache	1	FID of the index file
Extent cache	2	FID of the storage bitmap file
Quota cache	N/A	FID of the quota file

The cache flush lock is subordinate to the volume allocation lock. The file system uses the cache flush lock to interlock user file access to the three special system caches. Blocking ASTs are used to cause these caches to be flushed. The cache flush lock is normally held in a compatible mode (protected read). When everything is consistent within the cache, holding the lock in this mode allows the system to build cache entries. When an event occurs that requires the cache to be flushed, the lock mode is raised to concurrent write, which fires the blocking AST on the other nodes that eventually causes the caches to be flushed.

Specifically, the following actions are performed:

- The file system queues the AST control block (in the extent cache, the quota cache, and the file ID cache) to the process called the **cache server** (see Section 8.5.4). The AST parameter identifies the current volume and which cache needs to be flushed.
- In process context, the cache server process executes the appropriate control functions that cause the cache to be emptied.

- The file system releases the cache lock after the cache is flushed.
- The process that originally raised the cache lock proceeds.

For example, if a process opens the quota file for write, all the entries in the quota cache will be invalid. So when the quota file is opened for write, the cache lock is obtained, and the cache is flushed. The cache lock is also held as part of the opened file, which prevents the other nodes from taking out the lock again. This, in turn, prevents them from building cache entries, which they will not be allowed to do until the file is closed. Similar interlocking occurs for the FID cache and extent cache when the index file or storage bitmap file, respectively, is opened for write.

The extent cache also has to be flushed, generally when a process demands additional resources. For example, a user may request additional free space while extending a file. If all the remaining free space in the storage bitmap has been assigned to the local extent cache, and if there is still not enough space available to fill the user's request, the cache flush lock is raised as a signal to flush the caches on the other nodes of the cluster. As a result, the other caches are flushed, and the bitmap is then searched to see if the remaining space is adequate for the request.

Similarly, the file ID cache is flushed when no free file IDs are available on the system. For example, a user may want to create or extend a file. If all the valid file IDs in the cache have been used, the cache flush lock is raised to signal the other cluster members to flush their caches. After all the caches have been flushed, the index bitmap is scanned to find the free file IDs.

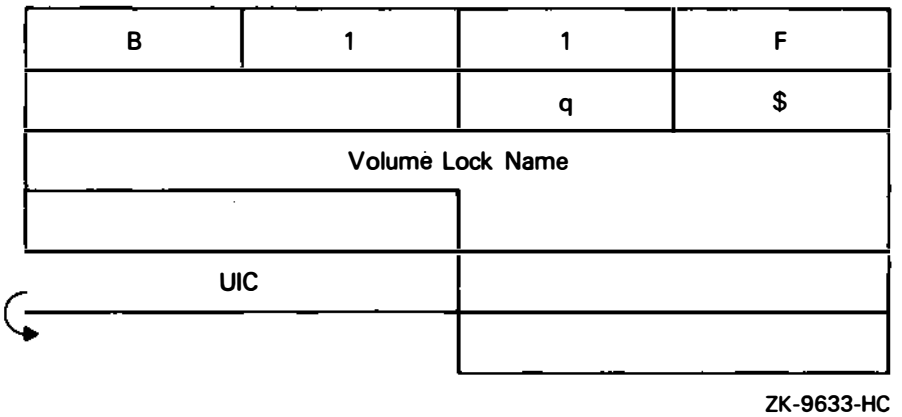
8.3.4 Quota Cache Lock

The **quota cache lock** is a top-level lock that controls access to quota cache entries. There is one quota cache lock for each quota cache entry (that is, for each quota cache record). It has the following form:

```
F11B$q<volume ID><UIC>
```

Figure 8–5 shows the in-memory representation of the quota cache lock.

Figure 8–5: Format of a Quota Cache Lock



The lock name is derived from the **F11B$q** prefix followed by the volume name and the UIC that is the individual name of the quota cache entry.

Each individual entry in the quota cache points to a quota cache lock. This lock is a top-level lock to promote locality of use because there are individual users on individual nodes in the cluster who are working with their individual disk quotas.

The value block contains the entire dynamic portion of the quota record; that is, the value block contains the authorized quota, which consists of the curent usage and other subsidiary pieces of information (such as the quota record number).

As with the other system-owned locks, a blocking AST is used on the quota cache lock in the following way:

- While a quota cache entry is validated on a particular CPU, the cache lock is held in protected write mode. That is, while the lock is held in an incompatible mode, the cache entry is marked valid. In this state, the current value of the quota can be maintained with no other copies. It has not been written to the quota file or anywhere else.
- When another node in the cluster wants to use that quota cache entry, it will enqueue for that lock in protected write mode.
- The blocking AST is fired on the node that is currently holding it. As a result, the following actions happen:
  - The lock is dequeued.
  - The value block is released.
  - The other node acquires the lock and the current cache contents.

In effect, the contents of cached entries are traded around the cluster by the lock manager.

8.3.5 Blocking Lock

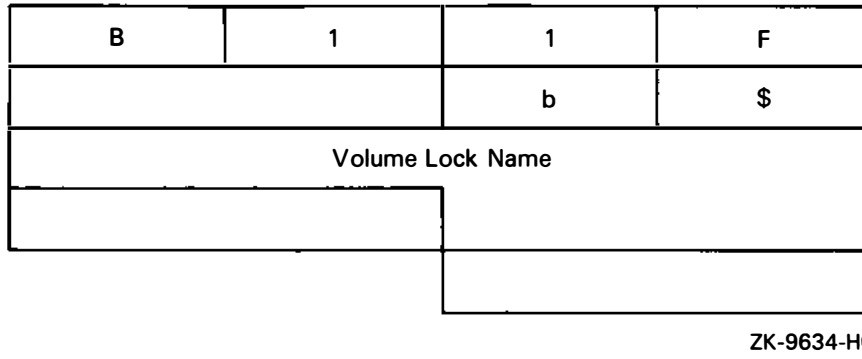
The **blocking lock** is the only lock that serves only one purpose; it is used to interlock all file activity on a volume when, for example, a rebuild operation is necessary.

The blocking lock has the following form:

```
F11B$b<volume ID>
```

Figure 8–6 shows the in-memory representation of the blocking lock.

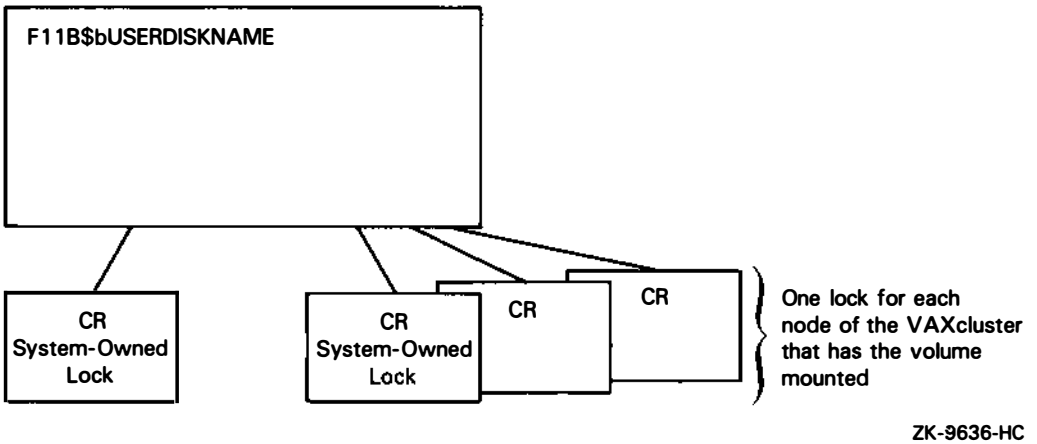
Figure 8–6: Format of a Blocking Lock



The lock name is derived from the **F11B$b** prefix followed by a 12-byte unique volume or volume set identifier.

The blocking lock is normally held in a compatible mode, which signals that normal file activity can proceed. Figure 8–7 shows the first stage in the life cycle of the blocking lock.

Figure 8–7: First Stage in the Life Cycle of the Blocking Lock

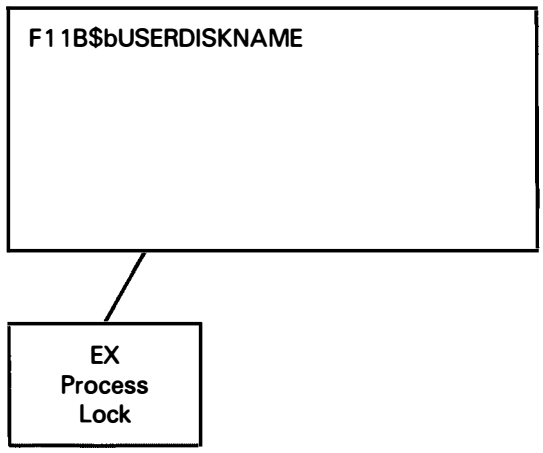


When a process wants to lock the volume, the following actions occur:

- The blocking lock is raised to an incompatible mode.
- As a result, blocking ASTs are fired to the other nodes in the cluster.
- If the file system is not active, these steps are taken:
  - The lock is dequeued.
  - The VCB is marked as blocked.

Figure 8–8 shows the second stage.

Figure 8–8: Second Stage in the Life Cycle of the Blocking Lock



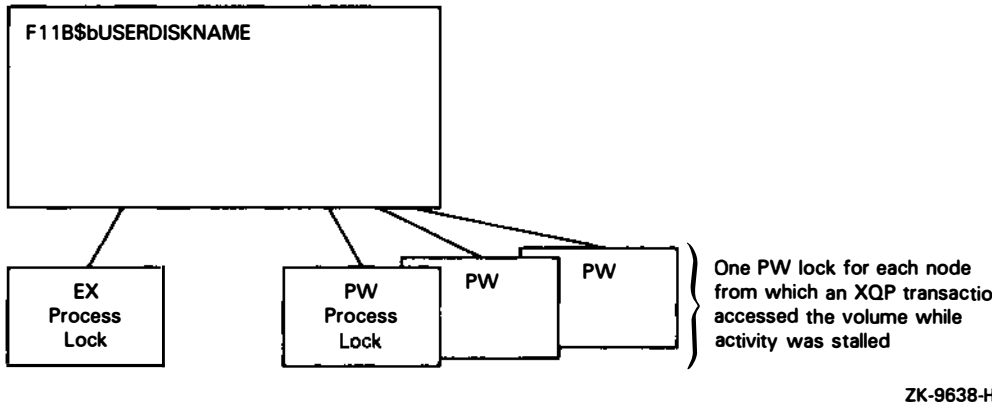
ZK-9637-HC

The following actions then occur:

- Subsequent file operations will try to re-enable the blocking lock by queuing for the lock in protected write mode.
- If the file system is active and the blocking lock is currently held, file operations will stall until the blocking lock is released.

Figure 8–9 shows the third stage in the life cycle of the blocking lock.

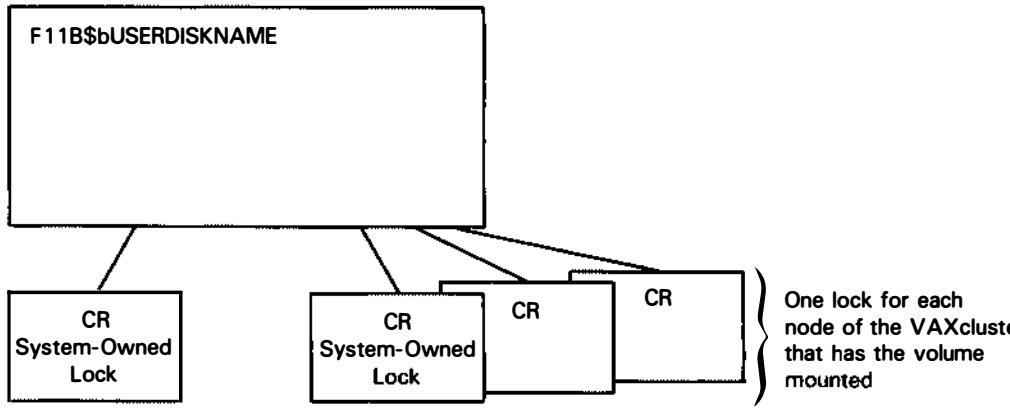
Figure 8–9: Third Stage in the Life Cycle of the Blocking Lock



When the blocking lock is released, the first process waiting for protected write gets the lock. It converts the lock back to a system-owned concurrent read lock and writes its lock ID in VCB$L_BLOCKID (or RVT$L_BLOCKID for a volume set). The remaining processes waiting for protected write mode execute in turn. Each one discovers that the blocking lock has already been rearmed by noting the nonzero value in VCB$L_BLOCKID (or RVT$L_BLOCKID), and so each simply dequeues its copy of the lock.

The volume and the volume allocation lock then return to their initial state. Figure 8–10 shows this fourth stage.

Figure 8–10: Fourth Stage in the Life Cycle of the Blocking Lock



8.4 Access Arbitration

The arbitration, or access, lock serves two purposes:

- To control file access by being held and mastered only on the node on which a file is opened
- To invalidate a stale file control block by means of a blocking AST
- To manage delayed file truncation

The routine ARBITRATE_ACCESS is called to arbitrate a new file access against existing accesses to a file. This routine first arbitrates the desired access against any pre-existing accesses on that node. Then it computes the correct mode for the access lock and attempts to take out the lock. (This information is contained in the FCB; the current access lock mode is stored in the FCB$B_ACCLKMODE field, and the lock ID in the FCB$L_ACCLKID field.)

ARBITRATE_ACCESS is normally called when controlled access to a file is required. For example, it is called under the following circumstances:

- To start up quota operations
- To determine if no write access to a directory has been requested in an explicit user open of a directory

  In this case, ARBITRATE_ACCESS is called to see if write access has been granted, but the access lock is returned to its original mode (implying a null lock for the requesting process).

- To open a file (except for explicit interlock ignore)
- To extend or truncate a file

  These functions are allowed to lower the access lock because they also hold the serial lock on the file, thus preventing any new accessors from interfering with the intended operation. The combination of the FCB reference counts and the LOCK_COUNT of the access lock indicates the other accessors of the file. Certain operations do not allow other users even though they are allowed by the normal access rules.

  The most obvious of these operations is truncation. A file cannot be truncated until it has no accessors other than the current process to ensure that an I/O is not in progress.
- To change the security classification of a file

ARBITRATE_ACCESS first does local arbitration by checking the desired file access against other accessors on the same node. (For example, a write access is checked against locks against writers, and so on.) If the system is not in a VAXcluster or if the volume is not cluster-accessible, only the local check is needed, and no access lock is taken.

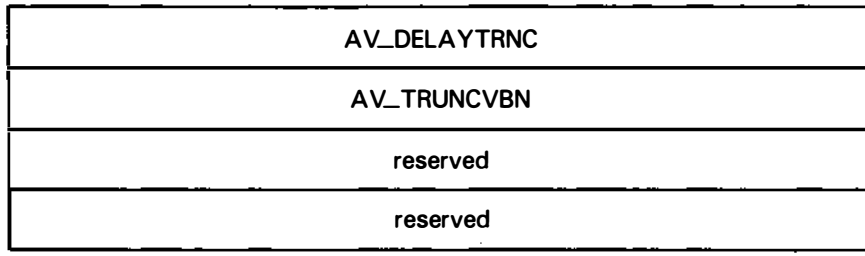
However, if the volume is cluster-accessible, the access lock is needed to arbitrate the access against other accessors elsewhere in the cluster. If this is the first access to the file on this node, the routine NEW_ACCESS_LOCK is called to take out a new access lock. Otherwise, CONV_ACCLOCK is called to convert the existing access lock to its new value.

When the file is deaccessed, CONV_ACCLOCK lowers the lock back to the supplied (previous) value. If the FCB reference count is zero, the access lock is dequeued because no other users on this node need it. It is called by the following routines:

- The MAKE_DEACCESS routine in CLENUP converts the lock when a process deaccesses the file. Using the updated reference counts in the FCB (FCB$W_WCNT and FCB$W_LCNT), new access control information is determined, which is then used to determine a new lock mode and lock value. If this lock value is lower than the current lock mode (or the node's FCB reference count drops to zero), CONV_ACCLOCK is called to convert the lock.
- DEACC_QFILE (in QUOTAUTIL) performs a similar computation when deaccessing the (nodewide) quota file.
- NUKE_HEAD_FCB, called by CLEANUP, MARK_DELETE, CLOSE_FILE, and UNHOOK_BFRD (when the directory index buffer block of a directory FCB is deleted) also requests a lock conversion to null mode to write out the value block. CONV_ACCLOCK also dequeues the lock if the reference count for the FCB is zero.

Figure 8–11 shows the value block of the access lock.

Figure 8–11: Arbitration Lock Value Block



ZK-9640-HC

8.4.1 Delayed Truncation

When a process accesses a file for writing, truncation is implicitly disallowed. The problem is that truncation depends on the ability to invalidate windows (WCBs) for all accessors. This is especially difficult because it would mean that the I/O that was in driver queues when the truncation was performed must be revoked.

The result is that truncation is only allowed by a single writer accessing the file. If other users have accessed the file for reading when the truncation is requested, the actual truncation is delayed or deferred until the last reader has deaccessed the file. This practice is very similar to the way in which a file is marked for deletion but is not really deleted until it is completely deaccessed.

The access lock is the mechanism used to determine when the last accessor has deaccessed the file. If a $GETLKI function returns a value of 1 for a count of locks on an access lock, this means that no user on any other node in the cluster has the file accessed. The routine LOCK_COUNT performs this function. In addition, the lock access count must be checked for other local accessors.

When a truncation is deferred, a flag is set in the value block of the access lock, along with the VBN to which the truncation was requested. The lock value block is how the information is passed to another node when the truncation occurs on one node but the last deaccessor is on another. Because this information is also recorded in the FCB, it is also necessary to mark the FCBs stale clusterwide.

If another writer accesses the file after the writer requesting the truncation has deaccessed the file, the delayed truncation operation is canceled. This is done by:

- Forcing the access lock to at least protected write mode

- Clearing the delayed truncation flags in the FCB (which implies the lock value block also)
- Converting the lock to that same mode (which will cause the value block to be written)

The first conversion to protected write mode is always possible immediately. However, if the delayed truncation flag is set, two conditions are indicated:

- A user requested truncation while the file was accessed in protected write mode (that is, no other writers were allowed). But because the file was successfully locked for writing, that other exclusive writer must have deaccessed the file.
- Some readers are still present (who must have allowed writers for the process to have succeeded in getting write access).

Thus, the highest mode requested in the cluster must be concurrent read mode, which allows the process to convert the lock to protected write mode.

When a file is deaccessed, the following actions are taken:

- Any requests for truncation are checked. If the process deaccessing the file is the only accessor, this check is made directly. Otherwise, the truncation validity checks are made and the values stored in the value block.
- The lock is upgraded to at least protected write mode. When the deaccess is actually finished, the subsequent lowering of the mode forces the value block to be written out.
- The following checks are made:
  - To see if the process had accessed the file for reading
  - To see if the process was the last one to deaccess the file

  If so, and delayed truncation was requested, the truncation is performed.

When a file is modified, a truncation request is allowed only if the process is the only accessor because MODIFY truncates the file at the time of the request. If the process has the file accessed, the following two checks can be made:

- The reference count in the FCB
- The lock count for the access lock

If the process does not have the file accessed, the access lock must be obtained (by ARBITRATE_ACCESS) specifying no readers. A check is then made to see if the process is the only accessor. CONV_ACCLOCK restores the original lock mode.

8.5 System Blocking Routines

There must be a reliable method for communicating within a cluster. Some resources are shared within a node, others with or by the cluster. For example, free space on the disk is shared by all cluster members because it is a clusterwide resource, as opposed to compute cycles, which is a per-node resource. These clusterwide resources must be coordinated.

System blocking routines are the mechanism by which the members of a VAXcluster system communicate certain clusterwide file system demands. For example, these demands may be requests for the members to release a quota lock for a volume or for the members to return all their free blocks from cache to the storage bitmap because one member cannot allocate enough free space.

System blocking routines are specific AST routines to handle the blocking ASTs from various system-owned locks. The system blocking ASTs are called in interrupt context. However, in response to some of these ASTs, locks must be dequeued or lowered. Calling the $DEQ or $ENQ service to do this requires execution in process context. For this purpose, the AST "borrows" the swapper process by queuing a kernel-mode AST to it. The system-owned lock is armed by the XQP before the first file system request is performed for a given volume as part of mounting the volume.

There are four system blocking routines:

XQP$BLOCK_ROUTINE	Blocks all activity on a particular volume. This routine receives the AST from the blocking lock, releases the lock, and marks the VCB as blocked. See Section 8.5.1 for more information.
XQP$REL_QUOTA	Releases the lock on the individual quota caches. See Section 8.5.2 for more information.
XQP$FCBSTALE	Invalidates the FCB when a file has been modified. This routine receives the blocking AST from the access lock and then sets the stale bit in the FCB. See Section 8.5.3 for more information.
XQP$UNLOCK_CACHE	Releases the contents of a cache. This routine receives the blocking AST for the cache flush lock and wakes up the file system's cache server process. See Section 8.5.4 for more information.

The system blocking routines are located in the SYS module SYSACPFDT. They are not FDT routines, but they need to reside in the nonpaged executive.

8.5.1 Volume Activity Blocking

The file system must be able to stall clusterwide requests to allow the storage and index bitmaps and the quota cache to be rebuilt while a volume is active and in use. Any changes that potentially modify these structures must be blocked.

Part of the blocking lock mechanism used to stall activity on a volume is the **XQP$BLOCK_ROUTINE**, which is a system blocking routine on the blocking lock (or activity blocking lock). The blocking lock is a system-owned lock held in concurrent read mode. It is armed by the XQP before the first file system request is performed for a given volume as part of mounting the volume.

One important time when it is necessary to stall activity on a volume is when a bitmap or the quota file is being rebuilt. This operation is a function of the REBUILD module in the MOUNT facility, and it occurs under the following conditions:

- When mounting a disk (MOUNT)
- When recovering cache contents for a volume that was improperly dismounted (SET VOLUME/REBUILD)
- When setting disk quotas for users and monitoring disk usage (System Management Utility (SYSMAN)[1])

Repairing errors in the file structure of a volume (with ANALYZE/DISK_STRUCTURE /REPAIR) is a similar function. Both the REBUILD module and ANALYZE /DISK_STRUCTURE/REPAIR use the ACP control lock volume function to prevent file creation, deletion, extension, and truncation activity while the volume is being rebuilt. In this way, the volume is prevented from being modified while the rebuild operation is in progress.

ANALYZE/DISK_STRUCTURE/NOREPAIR is the default. This command does not take out the volume lock, so false inconsistencies can be reported both because caches are not flushed and because the volume may change during the operation.

The file system controls when processing may be performed on a volume with two fields in the VCB (or the RVT for a volume set):

- The activity count field, VCB$W_ACTIVITY (or RVT$W_ACTIVITY), determines whether processing may be performed on the volume.
- The volume blocking lock field, VCB$L_BLOCKID (or RVT$L_BLOCKID), stores the lock ID of the blocking lock.

Either a nonzero value in the VCB$L_BLOCKID field or an odd (low bit set) value in the VCB$W_ACTIVITY field means that processing status is normal and may proceed.

---

[1] SYSMAN includes the functions of the Disk Quota Utility (DISKQUOTA), which operated as a standalone utility in VMS Version 4.6.

These fields are used in the XQP routine START_REQUEST. This routine is called from the DISPATCHER routine to check that processing status is normal before any serialization locks are permitted to be enqueued or before any file system processing which may affect or be affected by other file system processing is performed.

If the blocking lock currently exists, START_REQUEST adds 2 to the activity count (to preserve its odd value and to keep the low bit set) and returns. DISPATCHER then calls the appropriate routines to process the desired file system function.

If there is no blocking lock, START_REQUEST calls BLOCK_WAIT to wait out the lock (if any) and to take the blocking lock. The XQP$BLOCK_ROUTINE routine is specified as the blocking routine address, and the VCB address as the AST parameter, thus arming the blocking lock.

When the request is completed, the FINISH_REQUEST routine is called from DISPATCHER and decrements the activity count by 2, thus returning it to its original value.

When the ACP control function LOCK_VOL is performed, the routine TAKE_BLOCK_LOCK is called. This routine queues an exclusive mode lock for the F11B$b lock. This request is incompatible with the system-owned concurrent read lock, and on every node that holds that lock (normally, all the nodes that have the volume mounted), the lock manager calls the XQP$BLOCK_ROUTINE entry point at IPL$_SYNCH with the VCB address in R1.

The LOCK_VOL function stalls all activity by using the XQP$BLOCK_ROUTINE. The routine first checks the VCB or RVT activity count to see if the volume is active. It then decrements the previously odd activity count, making the value of the field even.

If the volume is not currently active, the following actions occur:

- If, as a result, the count becomes zero, the volume becomes idle, and further activity on the volume is blocked.
- After the VCB$L_BLOCKID field is cleared, a kernel-mode AST is queued to execute in the context of the swapper with the following information:
  - An AST parameter of the lock ID of the F11B$b lock
  - An AST entry point of the XQP$DEQBLOCKER routine

  The XQP$DEQBLOCKER routine causes the system-owned concurrent read mode blocking lock to be dequeued. The ACB that is used is contained in the VCB or the RVT. The ACB is part of the per-volume file system database in nonpaged pool, and it was set up by the file system when the lock with this blocking routine was armed.

However, if the volume is active (that is, if XQP$BLOCK_ROUTINE did not decrement the value in the VCB$W_ACTIVITY field to zero), the following actions occur:

- XQP$BLOCK_ROUTINE returns.
- The clear low bit of the activity count indicates that all further file system requests for the volume are blocked. Once the activity field is even (low bit clear), subsequent callers to START_REQUEST may not add to it. Instead, they must call the BLOCK_WAIT routine. This routine queues for the F11B$b lock in protected write mode, which will not be granted until the process that queued for it in exclusive mode dequeues that lock by doing an ACP control UNLK_VOL function.
- Further activity on the volume is stalled after the last request is completed (that is, when the process that called FINISH_REQUEST finally decrements the activity count to zero and dequeues the lock).

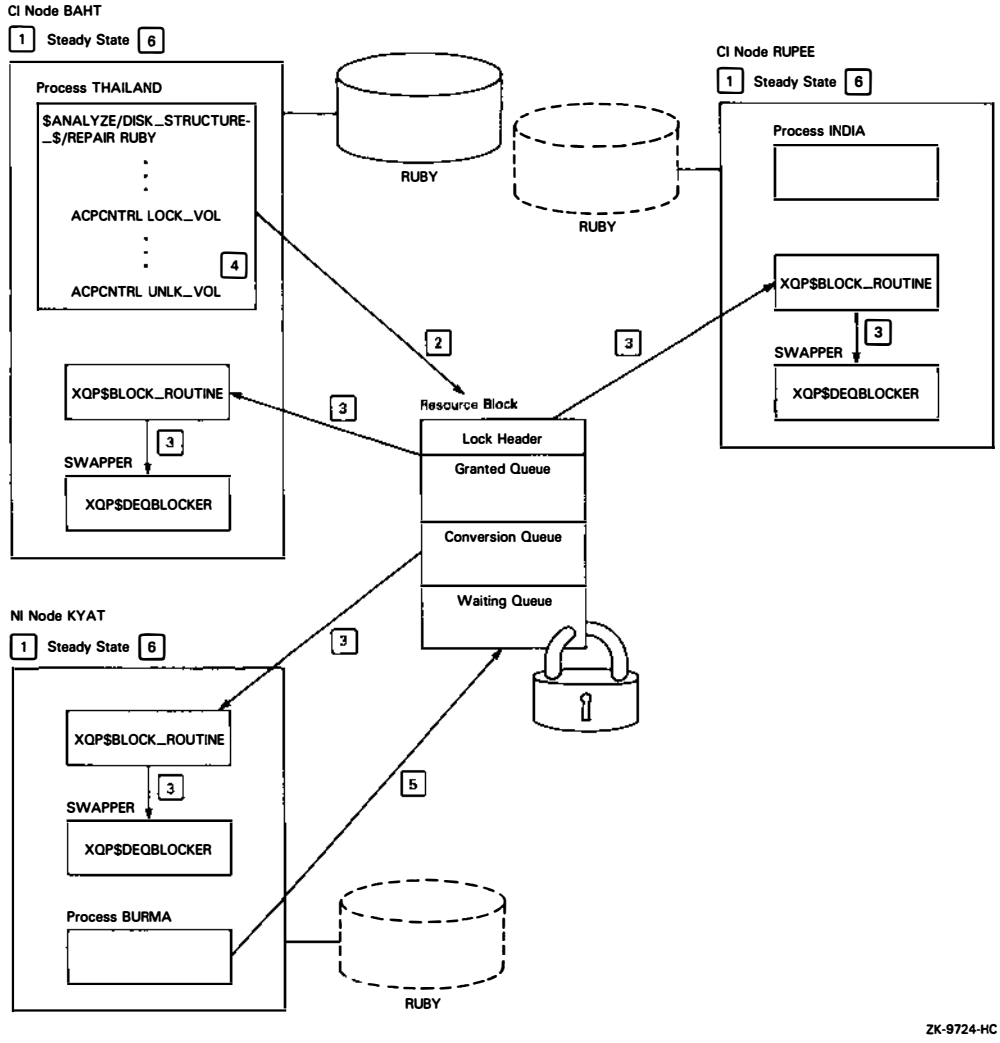
After all nodes of the VAXcluster have released their locks, and the volume is idle, the following steps are taken:

- TAKE_BLOCK_LOCK obtains the requested exclusive mode lock, and proceeds.
- The START_REQUEST and FINISH_REQUEST routines make their tests and changes at IPL$_SYNCH to interlock with the blocking routine correctly.
- When the exclusive mode F11B$b lock is dequeued, the first waiting process to get the protected write lock:
  - Converts that lock to the concurrent read mode system-owned lock
  - Sets the low bit of the ACTIVITY count, thus allowing events to proceed

  In fact, the F11B$b lock is initially armed in this fashion by the first function that calls START_REQUEST after the volume is mounted. Other users on the same node waiting for the lock find the blocking lock ID to be nonzero, so they must dequeue their (redundant) version of the lock.

Figure 8–12 shows how the XQP$BLOCK_ROUTINE system blocking routine is used in a VAXcluster.

Figure 8–12: XQP$BLOCK_ROUTINE Blocking Routine



The file system performs the following steps when handling clusterwide volume locking with the XQP$BLOCK_ROUTINE system blocking routine:

1. All nodes hold the volume allocation lock in concurrent read mode; this is the steady state mode.
2. Process THAILAND on node BAHT issues the command ANALYZE /DISK_STRUCTURE/REPAIR RUBY. The file system ACP control function LOCK_VOL is called to lock volume RUBY. LOCK_VOL raises the lock mode of the blocking lock to exclusive.

3. Because exclusive mode is incompatible with concurrent read, the swapper processes on nodes RUPEE and KYAT are delivered the blocking AST XQP$BLOCK_ROUTINE. This releases the concurrent read mode lock and immediately requeues the same lock with a BLKAST of XQP$BLOCK_ROUTINE. Process THAILAND on node BAHT is now granted a protected write mode lock on volume RUBY. This lock state transition is shown in Figure 8–13 through Figure 8–18.
4. THAILAND finishes the task and issues the ACP control function UNLK_VOL.
5. The blocking lock mode is demoted to concurrent read with the blocking AST. Process BURMA enqueues for the lock in protected write mode, and demotes to concurrent read mode when the lock is granted.
6. All nodes hold the blocking lock in concurrent read mode.

In addition, BLOCK_WAIT also returns the buffer credits it acquired while waiting for the blocking lock to be granted. This action is to avoid a deadlock caused by the cache server's not being able to flush caches due to lack of available buffers.

The process holding the exclusive F11B$b lock (via the LOCK_VOL function) and cache flush operations are allowed to proceed because both are necessary for the rebuild operation to work. The process holding the blocking lock is allowed to proceed because it has a nonzero value for BLOCK_LOCKID (one of the nonimpure XQP variables). However, any other file system activity (such as file creation, and so on) is prevented by the VCB$V_NOALLOC flag.

Figure 8–13 shows the initial lock state (the "steady state") of the resource block associated with volume RUBY during the ACP control lock volume function shown in Figure 8–12. All nodes hold the volume allocation lock in concurrent read mode.

Figure 8–13: Initial Lock State in a LOCK_VOL Function

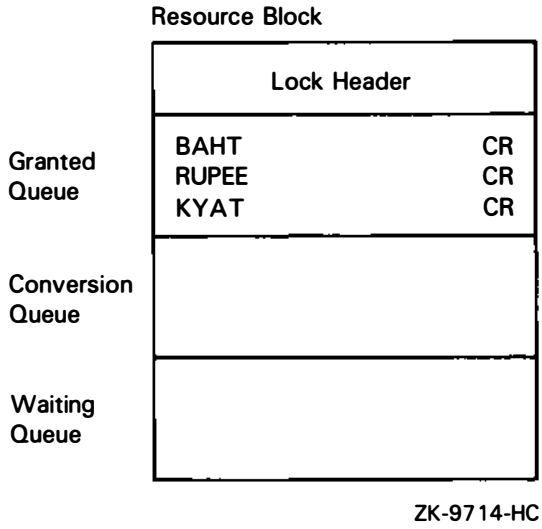


Figure 8–14 shows the lock state transition that happens when BAHT enqueues to convert its lock mode to exclusive because it requested a LOCK_VOL function. As a result, blocking ASTs are fired on the nodes RUPEE and KYAT. Because their modes (CR) are incompatible with EX, they are forced to drop the lock.

Figure 8–14: First Lock State Transition in a LOCK_VOL Function

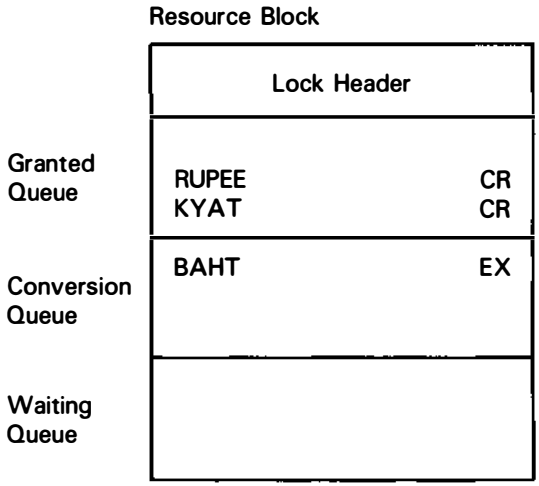
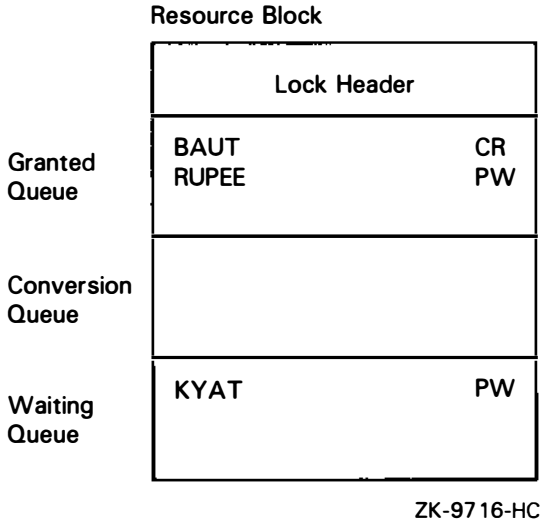


Figure 8–15 shows the lock state transition that happens when BAHT is granted an exclusive mode lock.

Figure 8–15: Second Lock State Transition in a LOCK_VOL Function



Nodes RUPEE and KYAT have no locks on the blocking lock until the next file system operation is attempted. Figure 8–16 shows the lock state transition that happens when, as a result of a file system operation, nodes RUPEE and KYAT re-establish the blocking lock. This action is started by enqueuing in protected write mode and then lowering to concurrent read (shown in Figure 8–18).

Figure 8–16: Third Lock State Transition in a LOCK_VOL Function

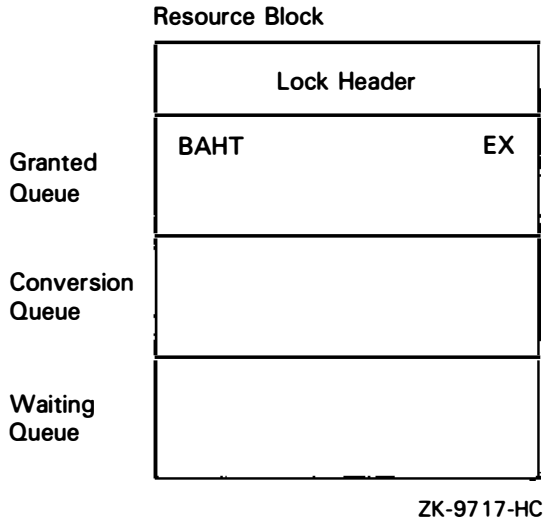


Figure 8–17 shows the lock state transition that happens after nodes RUPEE and KYAT have enqueued in protected write mode when BAHT unlocks the volume. BAHT's exclusive mode lock is demoted to concurrent read, and RUPEE is granted the blocking lock in protected write mode lock. KYAT remains in the waiting queue.

Figure 8–17: Fourth Lock State Transition in a LOCK_VOL Function

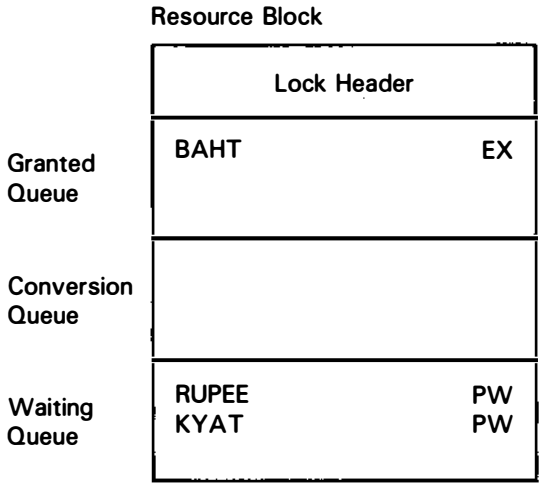


Figure 8–18 shows the lock state transition that happens as node KYAT is still trying to re-establish the blocking lock. RUPEE is demoted to concurrent read, and KYAT is finally granted the blocking lock in protected write mode.

Figure 8–18: Fifth Lock State Transition in a LOCK_VOL Function

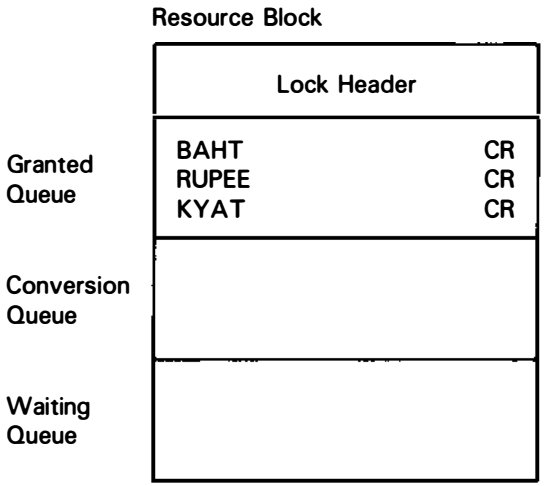
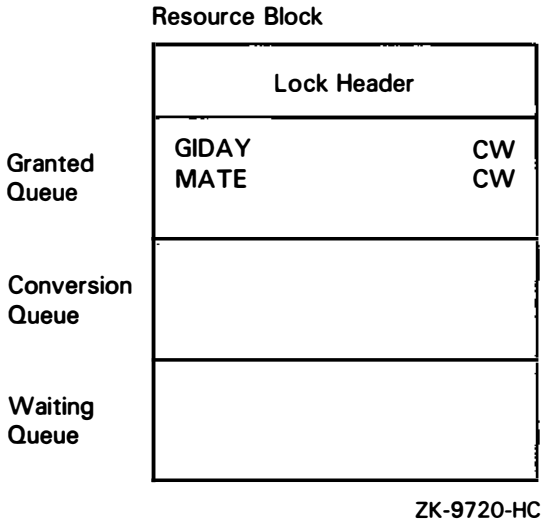


Figure 8–19 shows the final lock state (the "steady state") after KYAT has lowered its lock to concurrent read mode.

Figure 8–19: Final Lock State in a LOCK_VOL Function



The following functions also require the blocking lock:

- Modifying the quota usage of a process (QUOTAUTIL)
- Adding quota (the blocking lock is obtained in ACPCONTROL)
- Locking and unlocking ACP control functions (BLOCK_LOCKID is checked)

  A disadvantage to this approach, though, is that only one volume can be locked at a time, per-process. Also, failure of the process to unlock the volume prevents the volume from ever being unlocked.

The following functions do not require the blocking lock:

- Invoking a file system function (DISPATCHER)
- ACP control functions (with the exception of ADD_QUOTA)

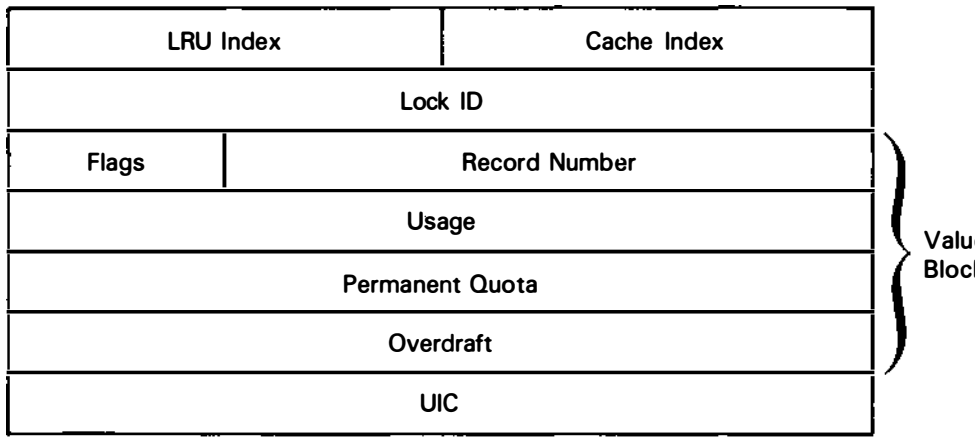
8.5.2 Dynamic Quota Cache Entry Lock Passing

Each quota record or entry in the quota cache has its own quota lock. When another node in the cluster wants to obtain the quota lock to modify a quota cache entry, the routine XQP$UNLOCK_QUOTA is called.

All the relevant information of the quota entry is packed into the value block of the lock, so it can be shared clusterwide.[1]

Figure 8–20 shows which fields of a quota cache entry make up the value block.

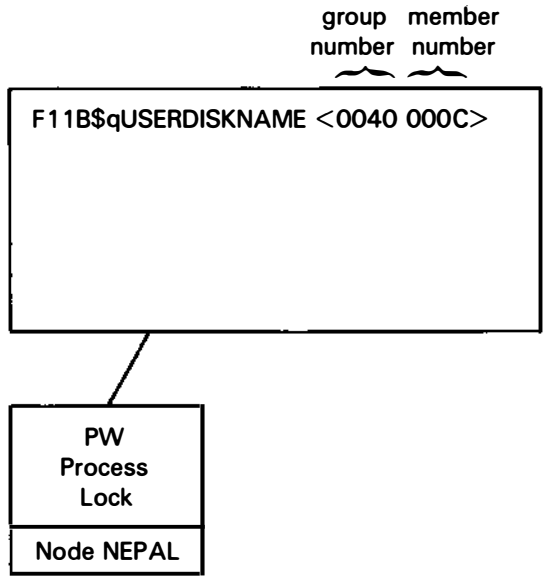
Figure 8–20: Quota Cache Entry Value Block



[1] If the value block is returned as invalid from the lock manager, the quota cache entry is likewise marked as invalid.

The quota lock is held in protected write mode. It is taken out when an entry is first cached. Figure 8–21 shows this first stage in the life cycle of the quota cache lock.

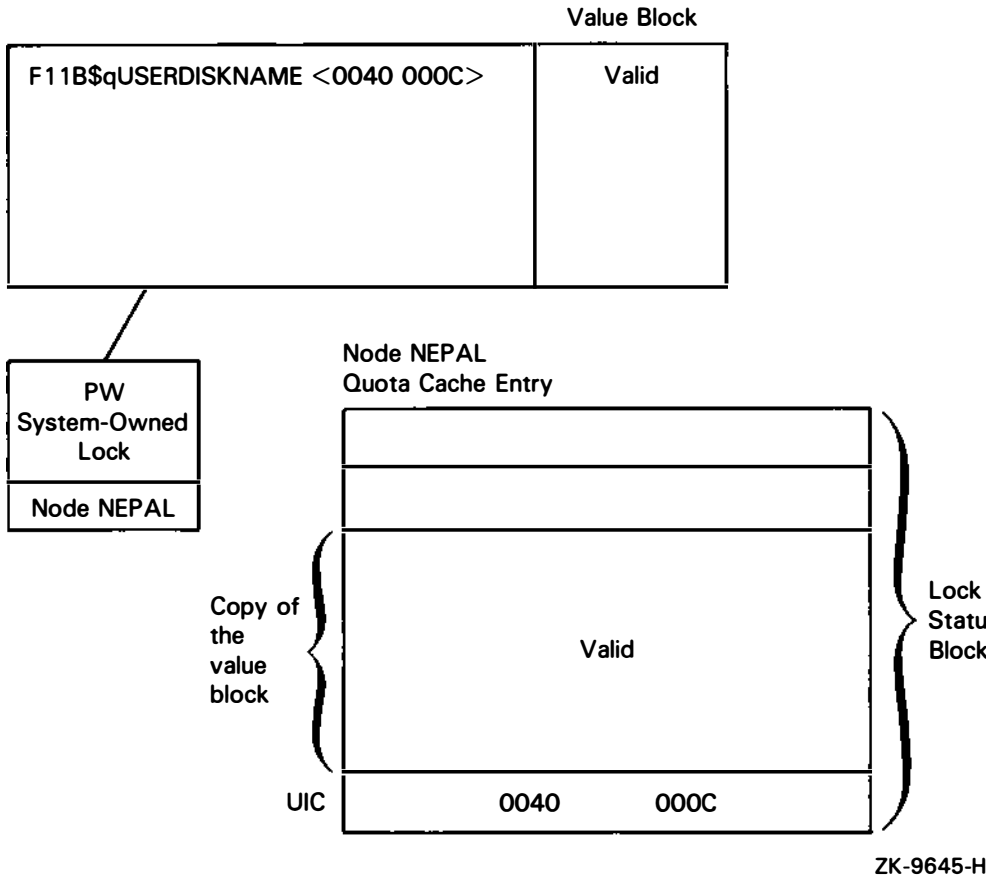
Figure 8–21: First Stage in the Quota Cache Lock Life Cycle



ZK-9644-HC

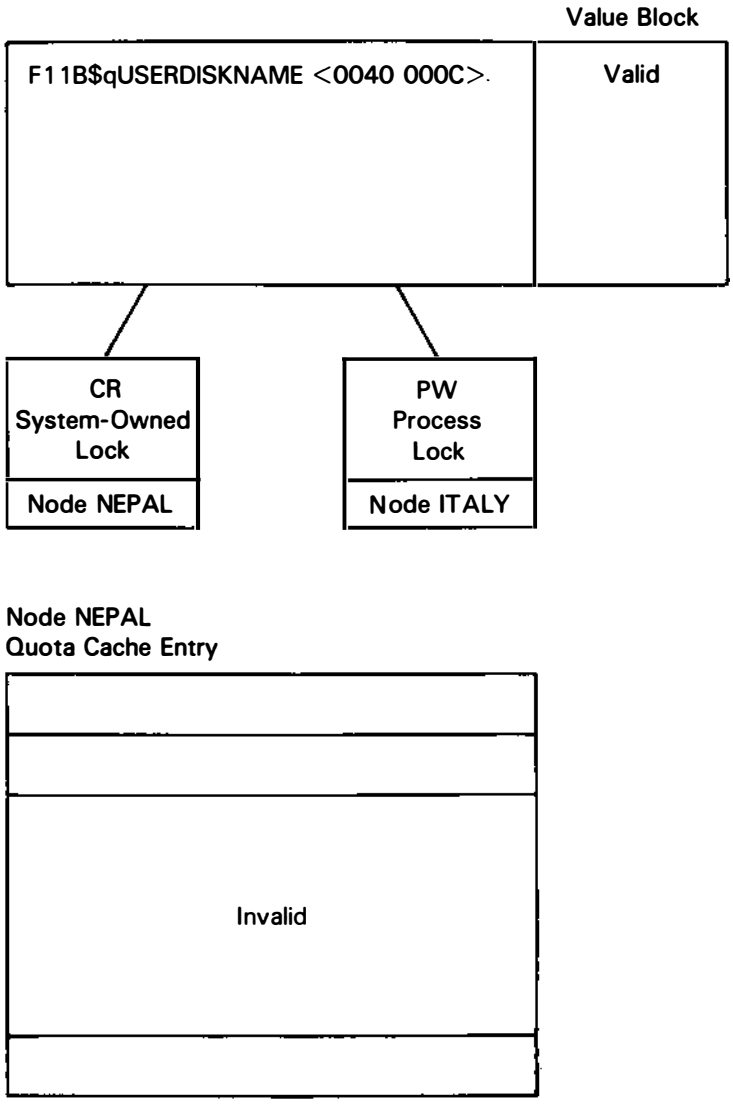
The lock is immediately converted to a system-owned protected write mode lock. It specifies the **XQP$REL_QUOTA** routine as the AST blocking routine. When a new entry is added to the quota cache (by SCAN_QUO_CACHE), the quota lock is obtained. When an entry is removed from the cache, either by explicit flush (in FLUSH_QUO_CACHE) or by LRU replacement, the lock is dequeued. Figure 8–22 shows the second stage.

Figure 8–22: Second Stage in the Quota Cache Lock Life Cycle



When another node queues for the F11B$q lock, the blocking routine XQP$REL_QUOTA is triggered, and an AST is sent to the swapper (in routine XQP$UNLOCK_QUOTA). If the quota entry is valid, the cache lock is demoted to concurrent read mode, which is compatible with the protected write mode that the other node has requested. This lock conversion updates the value block, which the other node picks up when it succeeds in getting its protected write mode lock. If the quota entry is not valid, the lock is dequeued entirely, and the entry marked vacant. Figure 8–23 shows the third stage.

Figure 8–23: Third Stage in the Quota Cache Lock Life Cycle



When a quota entry is removed from the quota file itself, the quota lock is requested in exclusive mode. This request forces any node that holds a quota lock on the quota entry to perform a cache flush of the entry. Figure 8–24 shows the fourth stage.

Figure 8–24: Fourth Stage in the Quota Cache Lock Life Cycle

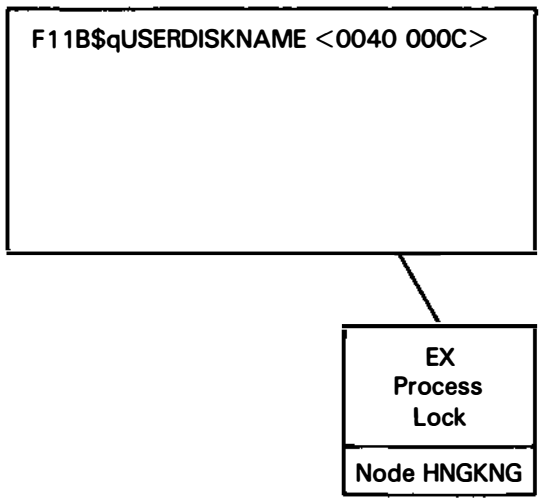
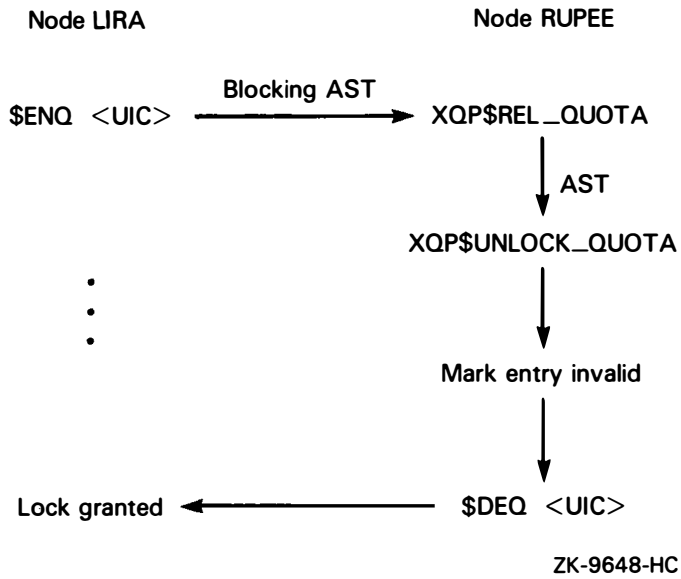


Figure 8–25 shows how the the quota cache is shared.

Figure 8–25: Quota Cache Sharing



8.5.3 FCB Invalidation

An FCB is an in-memory summary of a file's metadata, or the data about the file, rather than the data within the file. That is, when any action is performed upon a file, an FCB is first built from the header. For an accessed file, one FCB exists for each header in the file, and the FCB remains in nonpaged pool.

FCBs are queued from the VCB and can be found from either a window or from the file system context area. Most of the FCB is filled in from the file's header chain when the file is accessed. Thus, the information in the header and the FCB must be updated in a synchronized fashion. Since the file can be open for write access on more than one node in a cluster, multiple FCBs may exist for a file across the cluster. It is therefore critical to coordinate updates to the on-disk and distributed in-memory data.

Even if a file is accessed on one node, a process on another node may access the file header to change protection, to add extension headers, or to mark it for deletion. Because the XQP assumes the presence of an FCB without reading the header, a mechanism is needed to signal that an accessed file has been updated. Then the FCB chain can be rebuilt from the header chain only as needed.

A system blocking routine associated with the access lock is used for this purpose. The access lock is armed with the following pieces of information:

- The routine **XQP$FCBSTALE** as the blocking AST
- The primary FCB address as its parameter

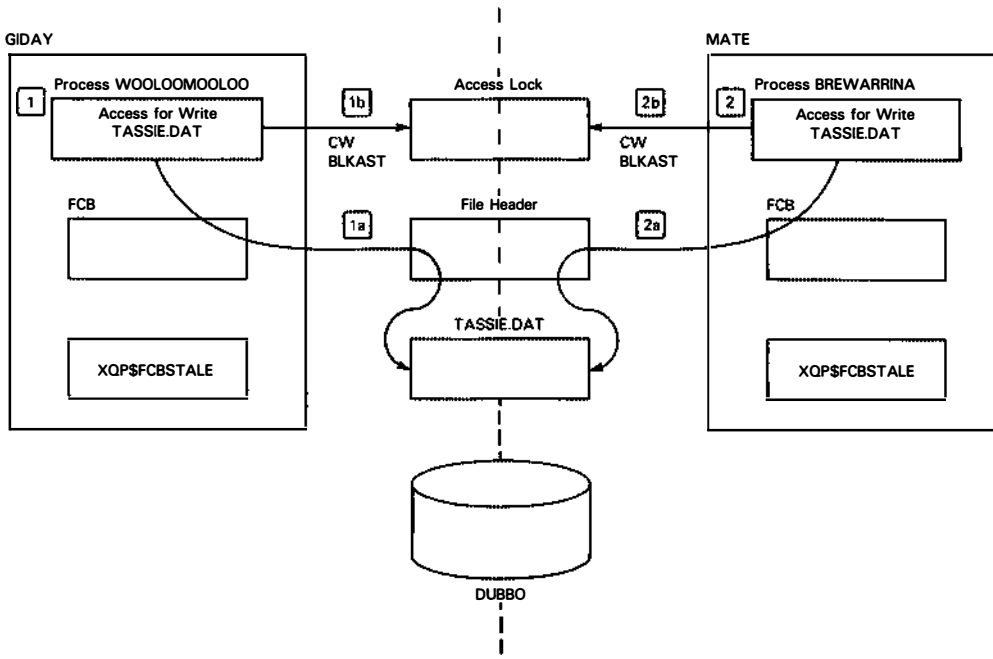
When a file header is modified in a way that changes the FCB contents, the XQP calls the routine MAKE_FCB_STALE. This action triggers the following sequence of events:

- MAKE_FCB_STALE queues for exclusive mode on the access lock.
- This action triggers the blocking routine.
- The AST routine sets the FCB$V_STALE flag in the FCB$W_STATUS field.
- Meanwhile, MAKE_FCB_STALE cancels the exclusive lock, which is generally never granted. In the rare case when it is, MAKE_FCB_STALE then calls CONV_ACC_LOCK to re-establish the original lock.

Figures 8–26, 8–28, and 8–30 comprise a series showing how an FCB is invalidated in a VAXcluster. After each of these figures, there is another figure (Figures 8–27, 8–29, and 8–31) showing the resource blocks associated with events in Figures 8–26, 8–28, and 8–30.

Figure 8–26 shows how a file can be shared for write access between two nodes of a VAXcluster.

Figure 8–26: First Stage of FCB Invalidation



1. Process WOOLOOMOOLOO issues a request to access the file TASSIE.DAT for write on node GIDAY, which generates an FCB on node GIDAY. As a result, the following actions occur:

   a. The file header is read.

   b. The access lock is taken out and held in concurrent write mode with a BLKAST of XQP$FCBSTALE and an ASTPRM of the address of the FCB.

2. The same file (TASSIE.DAT) is accessed for write on node MATE, which generates an FCB on that node. As a result, the following actions occur:

   a. The file header is read.

   b. The access lock is taken out and held in concurrent write mode with a BLKAST of XQP$FCBSTALE and an ASTPRM of the address of the FCB.

Figure 8–27 shows the initial state of the resource block associated with the file TASSIE.DAT before FCB invalidation. In this case, both nodes GIDAY and MATE hold the access lock in concurrent write.

Figure 8–27: Initial Lock State During FCB Invalidation

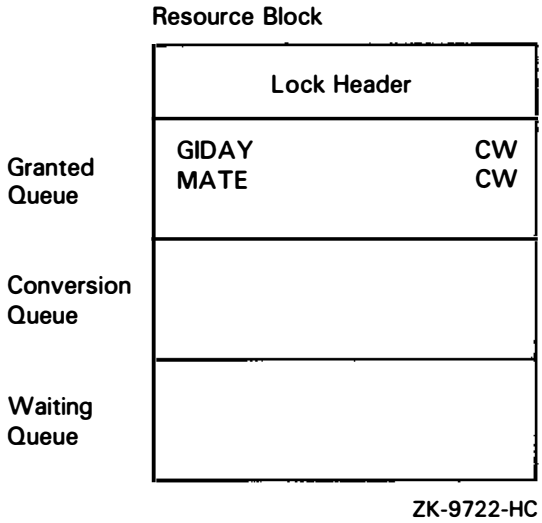
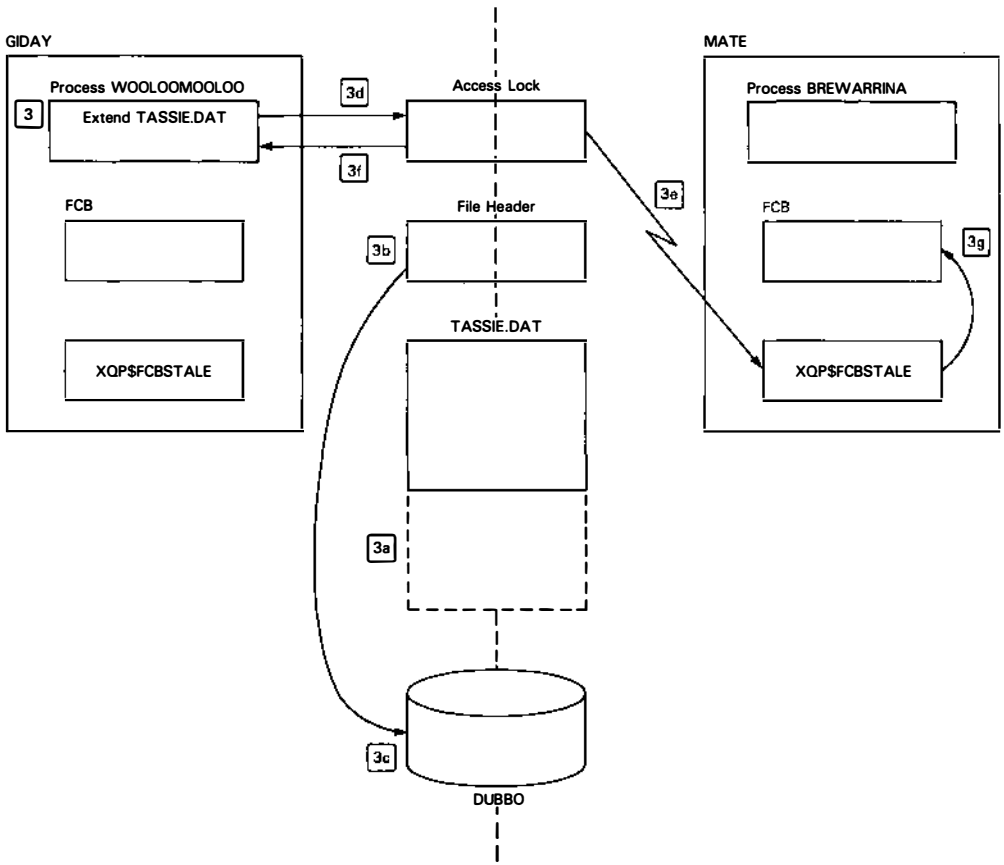


Figure 8–28 shows what happens if one node issues a request that would change the contents of the FCB.

Figure 8–28: Second Stage of FCB Invalidation



3. On node GIDAY, process WOOLOOMOOLOO issues a request to extend the file TASSIE.DAT. This causes the following actions to happen:
   a. TASSIE.DAT is extended.
   b. The file header for TASSIE.DAT is updated in memory.
   c. The primary header is flushed back to disk (the header is generally kept in memory) so that the other VAXcluster node has the means of updating the FCB from the reliable data on disk.

d. The access lock is requested in exclusive mode. Figure 8–29 shows the resource block after this request is enqueued.

e. The blocking AST (with a BLKAST of XQP$FCBSTALE and an ASTPRM of the address of the FCB) is fired.

f. The exclusive mode lock request is then canceled.

g. The FCB is marked stale on node MATE by the blocking routine XQP$FCBSTALE. XQP$FCBSTALE marks the appropriate bit (FCB$V_STALE) to indicate that the information in the FCB is no longer valid.

Figure 8–29 shows the lock state transition of the resource block after node GIDAY requests an exclusive mode lock in order to extend TASSIE.DAT. This is a new lock, not a conversion, and once requested, it is immediately dequeued.

Figure 8–29: Lock State Transition During FCB Invalidation

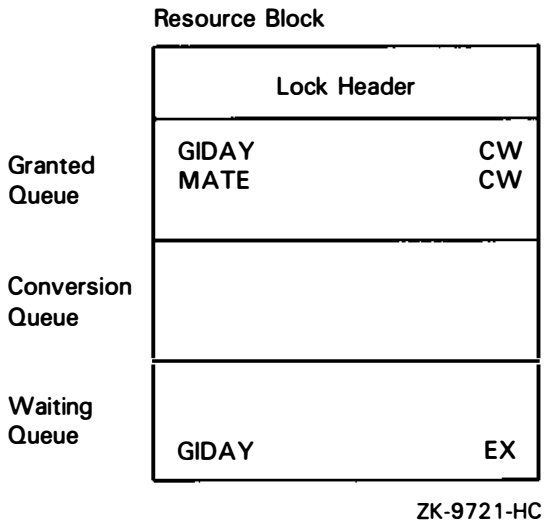
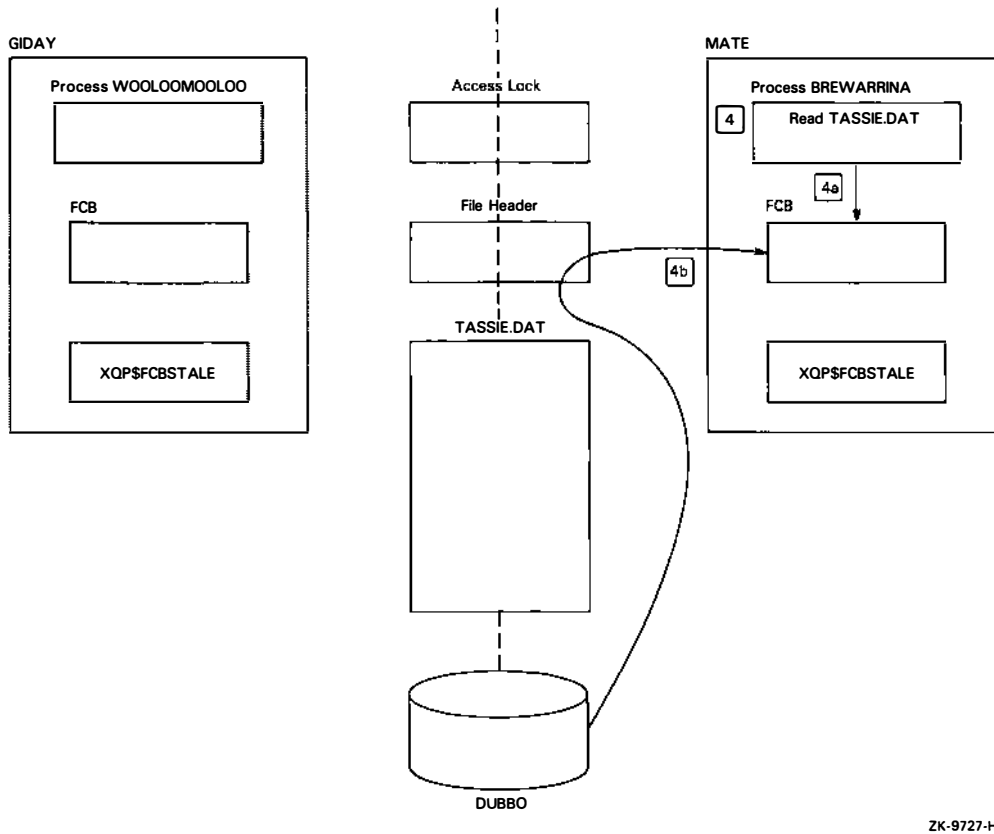


Figure 8–30 shows what happens when a node in a VAXcluster notices that the FCB$V_STALE bit is set.

Figure 8–30: Third Stage of FCB Invalidation



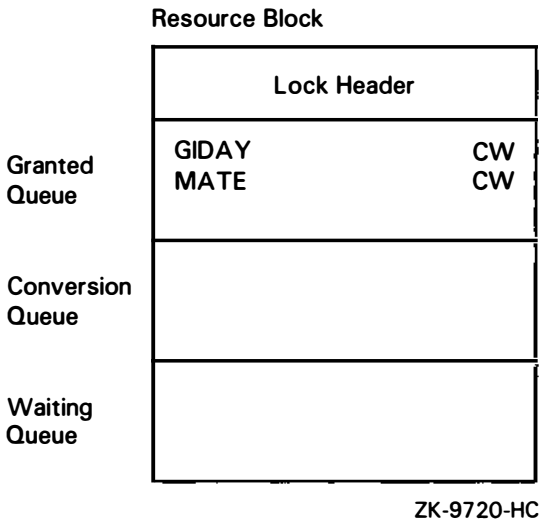
4. Process BREWARRINA on node MATE issues a read request. As a result, the following actions occur:

   a. Node MATE sees the FCB$V_STALE bit set.

   b. MATE updates its FCB by reading the header chain on disk. For a single-header file, this action involves two or more QIOs: a write on the node that marks the FCB, and a read function on each node that has an active interest in the file.

Figure 8–31 shows the final lock state of the resource block, where both nodes GIDAY and MATE once again hold the access lock in concurrent write mode.

Figure 8–31: Final Lock State During FCB Invalidation



The following functions require that the FCB be marked stale:

- Cleaning up after an operation

  This routine monitors the CLF_MAKEFCBSTALE flag, which is set by any routine that modifies a file header in such a way that would require rebuilding the FCBs. This flag is set by the following routines:

  - EXTEND and RWATTR, which modify protected attribute, UIC, class, file protection, and ACL information
  - TRUNCATE (although this function is performed by its callers)

    In the case of MODIFY, truncation is allowed only if there are no other accessors, so MAKE_FCB_STALE is not called. Likewise, MAKE_FCB_STALE is not called for DEACCESS because a delayed truncation implies that there are no accessors.

- Extending the quota file
- Extending or compressing a directory file (in SHUFFLE_DIR)
- Marking an FCB for deletion (in MARKDEL_FCB)

- Deleting a file (in DELETE)

  In this routine, the blocking AST is explicitly fired to signal that it has marked the file to be deleted when there are no other accessors.

- Deaccessing a file (in DEACCESS)

  This routine invalidates the local FCB chain if the access lock is held in null mode (all accessors on the node requested no lock). In this case, the FCBs are always questionable because blocking ASTs are not delivered to nodes holding null locks. CREATE_HEADER also invalidates the index file FCB for the same reason.

- Searching the FCB chain (in SEARCH_FCB)

  Like DEACCESS, this routine also invalidates the local FCB chain if the access lock is held in null mode.

- Accessing a file (ACCESS)
- Finding an entry in the quota file (SEARCH_QUOTA)

  This function may also require that the serialization lock be taken out on the quota file.

- Creating the file to be accessed (CREATE)
- Deaccessing a file (DEACCESS)
- Marking a file for delete (MARK_DELETE)
- Opening the file to be accessed (OPEN_FILE)
- Modifying a file (MODIFY)
- Connecting to and activating the quota file (CONN_QFILE)

When the FCB chain is found to be stale, the REBLD_PRIM_FCB routine is called to initialize a new FCB from the real file header and to rearm the blocking AST by converting the lock to the same mode. The BUILD_EXT_FCBS routine is then called to build the extension FCBs which might also have changed.

It is possible to open a file ignoring access interlocks by specifying FIB$V_NOLOCK in the access request. Such a file open is represented by a null mode access lock. Enqueuing an exclusive mode lock against the null lock does not cause the blocking AST to fire because null and exclusive mode are compatible. Because it is thus impossible to mark the FCB stale dynamically on such an access, an FCB opened with the FIB$V_NOLOCK bit is always assumed to be stale.

8.5.4 Cache Flushing

Some resources in a cluster can be shared but do not have to be cached. Likewise, some resources can be cached but cannot be shared. An example of a shared resource that can also be cached is the free space on the disk, which is controlled by the extent, or bitmap, cache. The presence of the extent cache (as well as the FID cache) implies that the index file and storage bitmaps do not actually reflect the true amount of free space available on the disk.

The extent cache is populated the first time that allocation or deallocation occurs on a volume after it is mounted. The amount of available storage on the disk is divided among the number of cluster members plus 1, and that amount is allotted to the extent cache on each cluster member. This technique guarantees that some space will be available to each VAXcluster member with some left over if the entire cluster fails.

For a variety of reasons, it may be necessary to flush the extent caches back to the bitmap. To do this, each node must be notified. This function is done with system blocking locks and the **cache server process**. This process is named **CACHE_SERVER**. Its process ID is stored in XQP$GL_FILESERVER, and its AST entry point is stored in XQP$GL_FILESERV_ENTRY. Each member of the cluster has its own cache server process.

If a cache is too small (if the number of cluster members is large), cache flush operations may occur frequently, causing performance degradation. In this case, smaller nodes or satellites may elect not to cache. A cache is volume-specific and is held under the volume lock. To be effective, a cache should represent at least 1000 blocks.

Flushing the extent cache can be triggered by the following conditions:

- When storage is allocated
- When the BITMAP.SYS file is accessed for write

Flushing the file number, or FID, cache can be triggered by the following conditions:

- Failing to find a free header (in the CREATE_HEADER routine)
- Accessing INDEXF.SYS for write

Basically, when these caches are in use, they hold the cache flush lock with the blocking routine **XQP$UNLOCK_CACHE** and an AST parameter that identifies the UCB with the cache type encoded in the low bits. The cache type is encoded as follows:

Value	Cache
1	FID cache
2	Extent cache
3	Quota cache

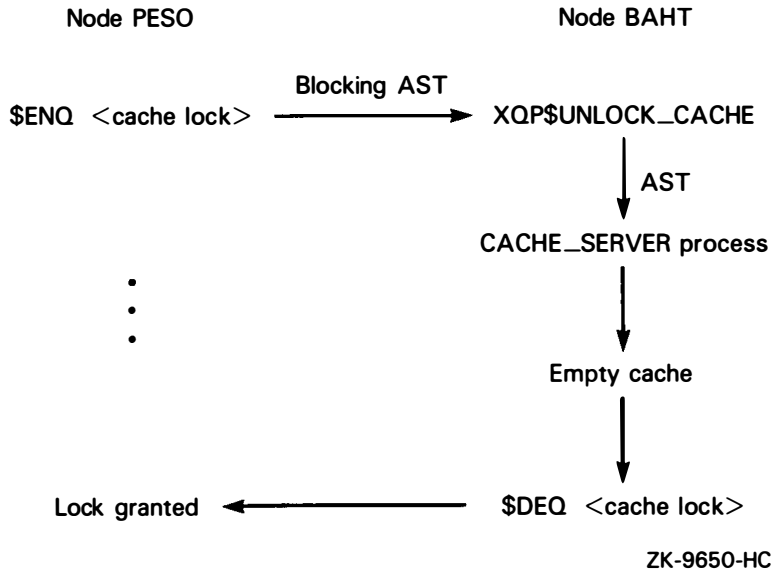
Normally, each node will take out a protected read mode lock on each cache. If any node needs specific processing (such as accessing one of the special files for write or requesting a cache flush elsewhere), a cache flush operation is requested as follows:

1. An incompatible lock in concurrent write mode is queued for the appropriate F11B$c lock (routine CACHE_LOCK).
2. The blocking routine in turn queues an AST for the cache server process with the appropriate AST parameter.
3. The cache server process, in turn, does a normal XQP call with a special ACP control function (CACHE_FLUSH) that flushes the correct cache.

   Flushing the cache also demotes the cache lock to null mode, indicating that the cache is no longer active. The process runs the image FILESERVER, which is a fully mapped program in P0 space. When it receives a blocking AST, the ACP control function is called. The cache flushing function needs a dedicated process to avoid deadlocks. Other processes may be in the middle of file operations themselves, or waiting for resources, or otherwise unable to execute the cache flush function.
4. The other nodes flush their caches.
5. After all cache locks are converted to null mode, the node which requested the cache flush operation gets the lock.

Figure 8–32 shows how a cache is flushed.

Figure 8–32: Flushing a Special Cache



The following conditions cause a system-owned lock in concurrent write mode to be taken out on a cache:

- When one of the special files (INDEXF, BITMAP, or QUOTA) is accessed for write. Other nodes must first release their protected read locks, so the process that requested the concurrent write mode lock performs the WAIT_FOR_AST and CONTINUE_THREAD routines. In addition, a concurrent write mode lock is taken out on the quota cache lock if the quota file is found to be write accessed at the time quotas are enabled.
- When an attempt to allocate a new file ID from the index file bitmap fails. When a new header is created, a process concurrent write lock is taken out on the cache lock. No blocking AST is associated with this lock, and it is dequeued as soon as it is obtained.
- When any attempt to allocate space from the storage bitmap fails. A process concurrent write mode lock is taken out on the cache lock. The process waits for the lock to ensure that all other nodes dequeued their protected read locks (that is, that they flushed their caches).

If a cache is found invalid (which is both the starting state for a cache and also the state after a cache flush), a protected read mode lock is requested on the cache lock. If the lock request fails, the cache is flagged as invalid.

In addition, the quota cache is also marked as needing to be flushed; otherwise, quota records will still be read into the cache and modified. The process does not wait for the lock because another node may be indefinitely holding a concurrent write mode lock (in the case that the file associated with the cache lock is accessed for write).

When a cache has been completely flushed, the cache lock is reduced to null mode to allow requesting nodes to continue with their lock requests.

When a volume in use is marked for dismounting, any special cache locks are dequeued (by CHECK_DISMOUNT). Similarly, a cache write lock is dequeued when a file is deaccessed (by MAKE_DEACCESS), and the quota cache lock is dequeued when the quota file is deaccessed (by DEACC_QFILE).

8.6 Cache Processing

Cache contents can be divided into the following three categories:

- Disk blocks (for example, the directory block cache)
- Parts of disk blocks (for example, the extent cache)
- Pointers to structures (directory index cache)

In a cluster, there are separate caches on each processor. The contents of a given buffer in the I/O cache on a specific processor will become invalid if that disk block is modified by the file system on another processor.

Because each buffer corresponds to some on-disk structure the file system manipulates, the reading and writing of those buffers must be serialized by one of the serialization locks discussed in Chapter 7. Those locks, and their value blocks, are the key to clusterwide buffer validation. One of the central aspects in the design of the XQP is clusterwide validation and invalidation of buffer contents.

8.6.1 Lock Value Blocks

Philosophically, a cluster may be thought of as a single system, and because the cache is a part of that system, the cache is split across members of the cluster. This distributed nature of the cache requires a lot of coordination. Dynamic coordination of the cache contents is performed by the following mechanisms:

- **Locks**—Both process-owned and system-owned. Because an ordering is associated with locks and implemented by the distributed lock manager, atomic operations are thus guaranteed.

- The **disk** itself—There are three ways the disk coordinates access to shared structures:
  - Through the **disk contents**

    When cache contents are found to be invalid, they are refreshed from the corresponding data on the disk.
  - Through **write-through caches**

    If a buffer is modified, it is written directly back to the disk.
  - Through **FCB invalidation**

    A set FCB$V_STALE bit causes the FCB to be flushed back to disk and the header chain to be rebuilt from the FCB.
- **Lock value blocks**—The distributed lock manager provides a general mechanism that associates lock client data with a lock. When a lock is passed around and ownership is gained, the value associated with the lock is passed along with it. The lock value block is the name given to the data carried around by the lock manager. Specifically, information about quotas is passed and maintained through lock value blocks.

The locks used to serialize access to a given disk block are also used to validate a cached copy of that disk block in a cluster. These are the F11B$v and F11B$s locks. The basic scheme of coordinating cache contents is to maintain a sequence number in the value block of the serialization locks (which are associated with specific buffers in the cache). This sequence number is incremented when any buffer associated with that lock is modified. All buffers associated with a given lock in a given cache retain a copy of the sequence number, which indicates the last time those buffers were used and valid. The sequence number changes every time the contents of the buffer change.

When a buffer is found in the cache by a later operation, the retained sequence number is compared to the current value from the serialization lock. If they match, no other processor has modified the associated disk block, and so the contents of the cached buffer are valid. However, if the retained sequence number and the current sequence number do not match, the contents of the cached buffer are invalid, and the buffer must be refreshed by reading the current contents from disk.

Different parts of different value blocks are used to validate different buffers. The following buffers are validated by the F11B$s file number serialization lock:

- File headers are validated by the FC_HDRSEQ field in the serialization lock for that file. A single sequence number is used to validate all headers for a given file; therefore, modifying just the primary header causes all cached headers for that file elsewhere to become invalid.

- Directory data blocks are validated by the FC_DATASEQ field in the serialization lock for a given directory file. All data blocks are validated by a single sequence number. However, a directory file header and its data blocks are validated by different parts of the same value block, so each can be independently modified without invalidating each other.

  Data blocks of any file opened by the internal OPEN_FILE routine are also validated by this field.

The actual fields in the value blocks are only referenced by the SERIAL_FILE and RELEASE_SERIAL_LOCK routines. They are referenced elsewhere by the LB_HDRSEQ and LB_DATASEQ vectors, and indexed by the lock index returned by SERIAL_FILE.

The following buffers are validated by the volume allocation lock:

- Storage bitmap blocks (BITMAP.SYS data blocks) use the VC_BITSEQ field.
- Index file bitmap blocks use the VC_IDXSEQ field.
- Quota file data blocks use the VC_QUOTASEQ field (bits 1 through 15 of the VC_FLAGS field).

The validation for buffers found in the cache is done by the FIND_BUFFER routine in the RDBLOK module. Modification of the sequence numbers is done by the WRITE_BLOCK routine. This is the only routine that writes modified buffers to disk.

When a node fails, the latest copy of the value block may be lost. The distributed lock manager then returns an SS$_VALNOTVALID warning status on $ENQ operations requesting the value block. In this case, the SERIAL_FILE and ALLOCATION_LOCK routines check for this status and increment all of the sequence number fields in the value block to force a cache miss. The SS$_VALNOTVALID condition is cleared by rewriting the value block.

8.6.2 Other Value Block Fields

The allocation lock value block also contains the fields VOLFREE, IBMAPVBN, SBMAPVBN, and IDXFILEOF. These fields are used as follows:

Field	Meaning
VOLFREE	Free volume block count. This field is passed around so that the last node to update the volume free block count can reflect that to other nodes. This count must be considered only approximate because a node can crash while it holds the value block, which would cause it not to reflect the true value. Assuming that the VOLFREE value is valid, EXTEND_INDEX uses this value in its algorithm to estimate the number of files to be created on the volume when it is deciding by how much to extend the index file. Likewise, this figure is used by SELECT_VOLUME to pick a volume for a file. The free figure is also used when deciding how many blocks to record in the extent cache. When a volume is unlocked, the unlocking node (which must also have been the locking node) has the only valid notion of free space. So LOCK_VOLUME saves the free space figure from the VCB and writes it back into the VCB under the allocation lock. (Acquiring the allocation lock updates the volume free count from some random value block.)
IBMAPVBN	Index file bitmap VBN. The index map VBN is maintained (by FILE_FID_CACHE and REMOVE_FILE_NUM) in the VCB as a starting point for header allocation (CREATE_HEADER). This value is used since it reflects the last place of interest in the index file map, a likely location for new headers. When file headers are allocated, the value is incremented to the index file map block from which the allocation was performed. If a FID with a lower value is returned to the map (from the FID cache), the value is decremented. When other systems fill their FID caches, their allocation will start with this block.
SBMAPVBN	Storage bitmap VBN. In a similar manner to the index map VBN, the storage map VBN is kept to record a starting point of interest in the storage map. It is updated only during storage map allocations. If the desired blocks are not found from this point to the end of the map, a scan is started from the beginning. This value may be reset to a lower value if the desired blocks are found lower in the map.

Field	Meaning
IDXFILEOF	Index file end-of-file. The index file EOF (set in CREATE_HEADER and EXTEND_INDEX) are passed around as an obvious limit to the header validation. The volume allocation lock must be taken out before the EOF can be obtained from the on-disk header.

8.6.3 Associating Locks with Buffers

The lock manager maintains two structures for a given lock:

- The **resource block**—Contains the resource name and the value block
- The **lock block**—Represents a specific lock on that resource and contains pointers to the granted, waiting, and conversion queues

The resource and lock blocks are created when the first lock is taken out on a given resource name. The resource block disappears when the last lock is dequeued.

The F11B$v and F11B$s locks used to serialize access to a specific disk block are also used to validate a cached copy of that disk block in a cluster. Because F11B$s locks are normally dequeued at the end of an operation, the resource block is deallocated and the value block is lost.

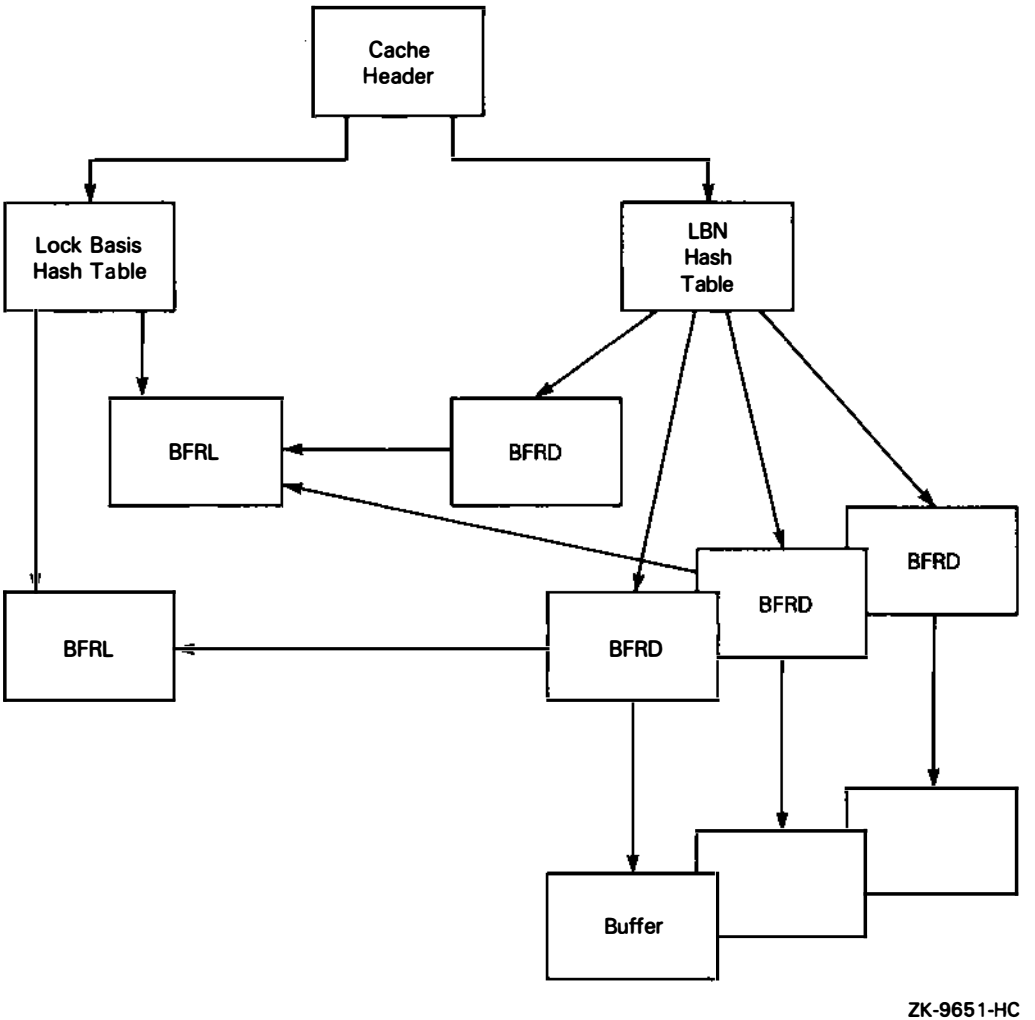
The problem with this locking approach is that just because a process is finished with a resource and does not want to keep track of it does not necessarily mean that the process is ready to return it altogether. However, the only way a buffer can remain in the cache is for a lock to be held on it.

Therefore, the concept of a system-owned lock was invented, which allows a granted lock to be converted so that it is no longer associated with a particular process. The resource it represents is available for a longer period of time than the process wanted to use it. To keep track of what buffers will remain in the cache, the cache is handled in a **least recently used**, or LRU, fashion.

When a buffer is in the cache, it must have a null-mode, system-owned lock associated with it. A **buffer lock descriptor** or **BFRL** is used to keep track of these locks. Multiple buffers may have the same lock basis, so many **buffer descriptors** or **BFRDs** may point to the same BFRL. The BFRL contains a reference count of the number of BFRDs so the file system knows when the lock can be completely dequeued.

Figure 8–33 shows the relationship between BFRLs and BFRDs in the file system cache structure.

Figure 8–33: Buffer Descriptors and Buffer Lock Descriptors



8.6.4 Cache Invalidation

Although most locks protect a single resource, caching is a case where locks are used to guarantee the validity of the cache contents. The major technique for protecting the cache from corruption and ensuring its validity is the lock value block.

A problem with this locking approach is that the sequence number is transmitted from system to system by the value block associated with the lock protecting a set of buffers. Because many buffers may be associated with a particular BFRL (for example, there might be ten storage bitmap blocks for the same volume in the cache), their sequence numbers must be synchronously updated.

If one of the buffers is modified by another operation on that system, both the BFRD$L_SEQNUM field and the appropriate value block field will be updated. However, other buffers held under the same lock but not referenced in this operation need special handling; otherwise, their sequence numbers would not be updated, and they would appear to be invalid when they were subsequently referenced.

This problem was solved with additional logic introduced in Version 5.2. After the buffers used in the current operation are released, all buffer descriptors held under the lock are checked. Those BFRDs that are currently valid are marked with the udpated sequence number to keep them valid.

Releasing a serialization lock and potentially converting it to a system-owned lock is done by two routines working together: RELEASE_SERIAL_LOCK and RELEASE_LOCKBASIS.

RELEASE_SERIAL_LOCK calls RELEASE_LOCKBASIS to perform the following actions:

- Scan the in-process list of buffers.
- Search for a specific lock.
- Associate a BFRL with the buffers if one does not already exist.
- Scan the BFRDs associated with the BFRL. The buffer is still valid if the two following conditions apply to the buffer sequence number in each buffer:
  - Is equal to or greater than the sequence number read from the lock value block when the lock was taken
  - Is less than or equal to the current sequence number

  If both these conditions exist, the buffer sequence number receives the current sequence number.
- Return to RELEASE_SERIAL_LOCK with the appropriate status if a new BFRL has been created.

When RELEASE_LOCKBASIS returns with the status indicating that a new BFRL has been created, RELEASE_SERIAL_LOCK performs the following actions:

- Converts the serialization lock to a system-owned lock.
- Stores the lock ID in the BFRL.

Otherwise, the serialization lock is simply dequeued. Because the cache serialization interlock is held during this scan, the process cannot be stalled. For that reason, all modified buffers must have been written before RELEASE_SERIAL_LOCK is called. Modified buffers may be written by making explicit calls to WRITE_BLOCK for individual buffers, or to the WRITE_DIRTY routine to scan the lists.

For a nonclustered system or disks that are not cluster-accessible, null locks are not associated with buffers because they are not necessary.

The allocation lock protects the storage bitmap and index file map blocks, so they must be written to disk and released before the allocation lock can be released. ALLOCATION_UNLOCK performs this function.

The DELETE_FILE routine, when purging the buffers for extension headers, fabricates a serial lock on the extension header file ID as a basis for purging the buffers. The buffer purging is done immediately instead of waiting for the normal cleanup procedure later to prevent another process from using these file IDs as primary headers and reading the buffers.

8.6.5 Directory Index Cache

The directory index cache occupies the fourth pool in the buffer cache. This cache does not contain buffers, though, but rather an index into a given directory file, constructed as the directory is processed.

There are actually two file system directory caches:

- A **block cache** containing a portion of data from a directory file.
- A **directory index cache** containing information about the contents of a specific directory file. Entries in this cache may even point to directory file blocks that are no longer present in the block cache.

The main purpose of the directory index cache is to serve as an index into the contents of the sequential files that serve as the directory file. This index allows the file system to locate directory entries directly rather than having to scan the directory file from the beginning.

A directory index cache can outlive access to a directory when a file is accessed through a directory path. Likewise, a directory index cache entry (along with the FCB) can outlive access to a file. Thus, when the directory file is opened again, it is still in the directory index cache, and the directory FCB still exists.

The directory index cache is both a CPU and an I/O saver. It requires the file system to reference fewer blocks in the directory to find an entry, which requires fewer cache buffers to hold the blocks, fewer I/O operations to read them, and less CPU time to search them.

RMS has a related cache—the **directory pathname cache**. (See Section 8.6.6.)

The directory index cache is managed by the buffer cache code because it essentially has the same clusterwide content validation problems that other caches do. One of its purposes is predicting the cyclic usage of caches.

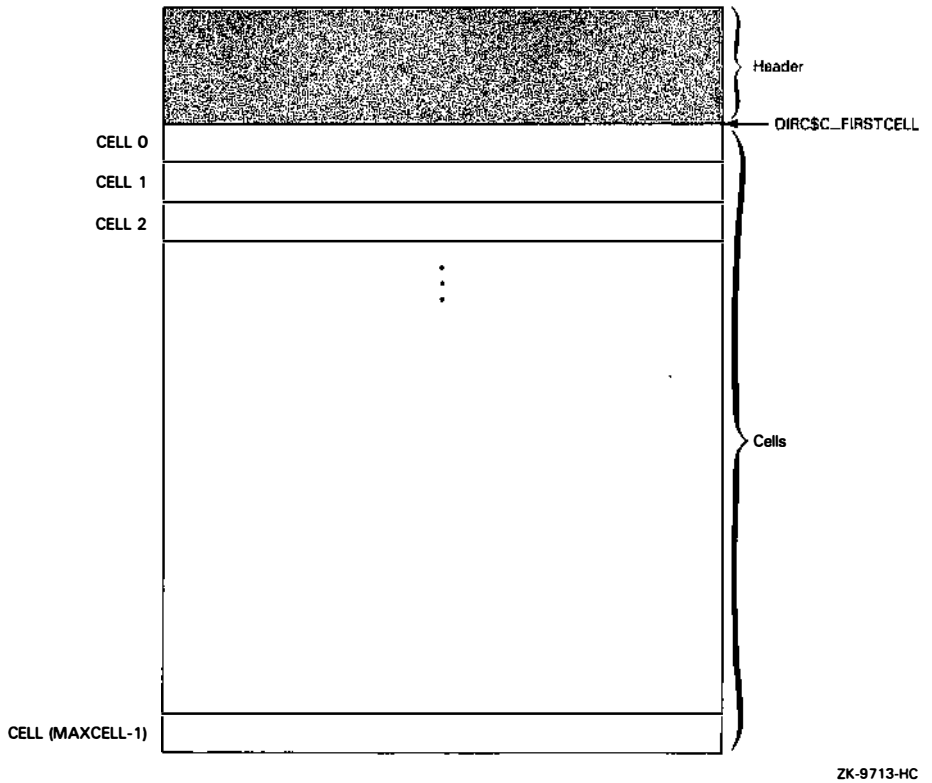
A directory index block has a small header area followed by about 30 15-byte cells. Each cell represents the highest record found in the corresponding directory data block. This scheme allows every block to have an entry for directory files smaller than thirty blocks; every other block for directories between 30 and 60, and so on. The cell size is set to 15 characters for small directories and is reduced for larger directories to allow more cells. The total index is limited to 512 bytes for each directory. A size of 15 bytes was picked because MAIL$800... files that are about a day apart in creation time vary in the fourteenth or fifteenth character.

Instead of being located by hashing on the LBN, a directory index block is pointed to by the FCB$L_DIRINDX field in the directory FCB (protected by IPL$_SCHED). The BFRD$L_LBN field points back to the directory FCB. If the FCB has no corresponding DIRINDX block, one is removed from the list for the directory index pool and linked to the FCB. Otherwise, the block is used.

The block is validated from the LB_HDRSEQ and LB_DATASEQ values for the directory. Only one directory index block is allowed to be used by the process (because the XQP only works with a single-parent directory in an operation).

Figure 8–34 shows the location of the directory index cache pool.

Figure 8–34: Directory Index Cache Pool



Some important routines that work with the directory FCB are as follows:

Routine	Meaning
MAKE_DIRINDX	Validates a directory FCB. It is located in RDBLOK and called from DIR_ACCESS in DIRACC.
KILL_DINDX	Breaks the linkage between the FCB and the directory index block. ERR_CLEANUP calls KILL_DINDX when it needs to delete a directory with a corresponding directory index block. Likewise, when MARK_DELETE deletes a file (when the reference count goes to zero), it calls KILL_DINDX before purging the FCBs.

Routine	Meaning
	KILL_DINDX also invalidates the directory index block pointed to by a particular FCB. It calls UNHOOK_BFRD to unhook the buffer descriptor for a directory index block to break the link to the FCB. This is done when the BFRD pops to the top of the LRU list and is being used for a new directory. Other routines which also unhook the BFRD are KILL_BUFFERS and KILL_CACHE.
SET_DIRINDX	Tries to maintain FCBs for popular directories and to keep the association of the FCB to the directory index block. It is called from CLEANUP and CLOSE_FILE. If the caller of SET_DIRINDX finds that the reference count for the FCB goes to zero, the FCB would normally be deleted. However, if a directory index block exists, the FCB$V_DIR bit is set.
DIR_ACCESS	Requests the creation of a directory index block. Finding a valid block indicates a valid FCB (because of the checks in MAKE_DIRINDX). Otherwise, the header is read, and MAKE_DIRINDX is called. If the directory is not really a directory, KILL_DINDX is called to remove the invalid directory index block. Likewise, clearing the directory bit (by WRITE_ATTRIB) also calls KILL_DINDX.
UPDATE_INDX	Builds the directory index block as it is scanned. This routine is called by ENTER and DIR_SCAN. DIR_SCAN uses the block it found to save work on subsequent scans.
ZERO_IDX	Updates the directory's header. The routine is located in CLENUP and is called by SHUFFLE_DIR. It increments the FCB$W_DIRSEQ field and sets the corresponding INUSE value in the directory index block to zero because the block layout is not different. FCB$W_DIRSEQ is updated under the following conditions: • When a directory is directly accessed • When SHUFFLE_DIR rearranges the directory contents • After a directory is accessed but a valid directory index block is not located

For more information on the RMS directory cache, see Section 8.6.6.

8.6.6 RMS Directory Pathname Cache

The RMS directory cache is a list of directory names and file IDs that RMS has recently processed. Its purpose is to save RMS from having to call the XQP when repeatedly stepping down levels of a subdirectory tree. However, the XQP must still be called to step down the tree under the following conditions:

- When the directory tree is initially referenced
- If the directory structure is changed (for example, during a delete or rename operation, or when the volume is dismounted)

To ensure that the RMS directory cache is valid and that it is synchronized with the XQP cache, the sequence number UCB$W_DIRSEQ is stored with the directory entries cached by RMS. Whenever the file system changes the directory structure, UPDATE_DIRSEQ (in CHKDMO) is called. The following directory operations udpate the sequence number:

- Superseding a directory (ENTER)
- Removing a directory name (REMOVE)
- Clearing the directory bit (RWATTR)
- Mounting the volume

The volume lock is used for RMS cache invalidation in the following way:

1. When the volume is first mounted, the volume lock is converted to concurrent read mode by RM$ARM_DIRCACHE (in RM0SETDID). This routine specifies RM$DIRCACHE_BLKAST (in the SYS module RMSRESET) as the blocking AST routine.
2. When the XQP invalidates the RMS directory cache, it temporarily enqueues the volume lock for exclusive access (in QEX_N_CANCEL). This action triggers the blocking AST routine RM$DIRCACHE_BLKAST.

   This routine refers to the high-order bit of the UCB$W_DIRSEQ field as the armed bit, indicating that blocking AST of the volume lock is armed. When this blocking AST routine fires, the UCB$W_DIRSEQ field is incremented, and the armed bit is cleared.
3. When the next RMS user references a directory file on the device whose UCB$W_DIRSEQ field was incremented, RMS compares the sequence number for the RMS directory cache entry with the sequence number in the UCB and finds that they are not equal. RMS then flushes all RMS directory cache entries for this device. RMS also calls RM$ARM_DIRCACHE again to rearm the blocking AST of the volume lock and to set the armed bit.
4. When the volume is dismounted, the armed bit is cleared when the lock (and therefore the blocking AST) is disarmed (by CHECK_DISMOUNT).

8.6.7 User Invalidation of Cached Buffers

Because all volume structures that the file system uses to maintain the volume are themselves files, it is possible for random users to access those files and to read and write the blocks in them. For example, anyone with write access to a directory can open the directory file and write random data into its data blocks. Or with appropriate privileges, a user could open BITMAP.SYS and rewrite it. A disk rebuild operation does this, for example.

The problem is how to invalidate cached copies of those blocks that may be in the buffer cache.

The solution is to trap all write virtual requests in the file system. This is done by setting the WCB$V_WRITE_TURN flag in the WCB of any write-accessed directory file (by ACCESS), or the index file, or the storage bitmap file (by MAKE_ACCESS).

When a write virtual function is performed on one of those files, the QIO FDT routine forces a window turn. The appropriate cache is flushed under the allocation lock when one of these files is initially accessed for writing. The cache flush lock is then taken out.

The XQP takes out the appropriate serialization lock (if it is not already held) to validate that buffer and increment the appropriate field in the value block (in the READ_WRITEVB routine). In many cases, a program that accesses file structure files for write should first take out the blocking lock to block other activity on the volume. Patching elements of the file structure is usually done in conjunction with other analysis of the file structure; such analysis will only be valid if the file structure cannot change.

The cache flush lock is released in MAKE_DEACCESS.

If the node is not in a cluster, the KILL_BUFFERS routine is called by READ_WRITEVB to scan the cache and invalidate the correct buffers.

Index

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z