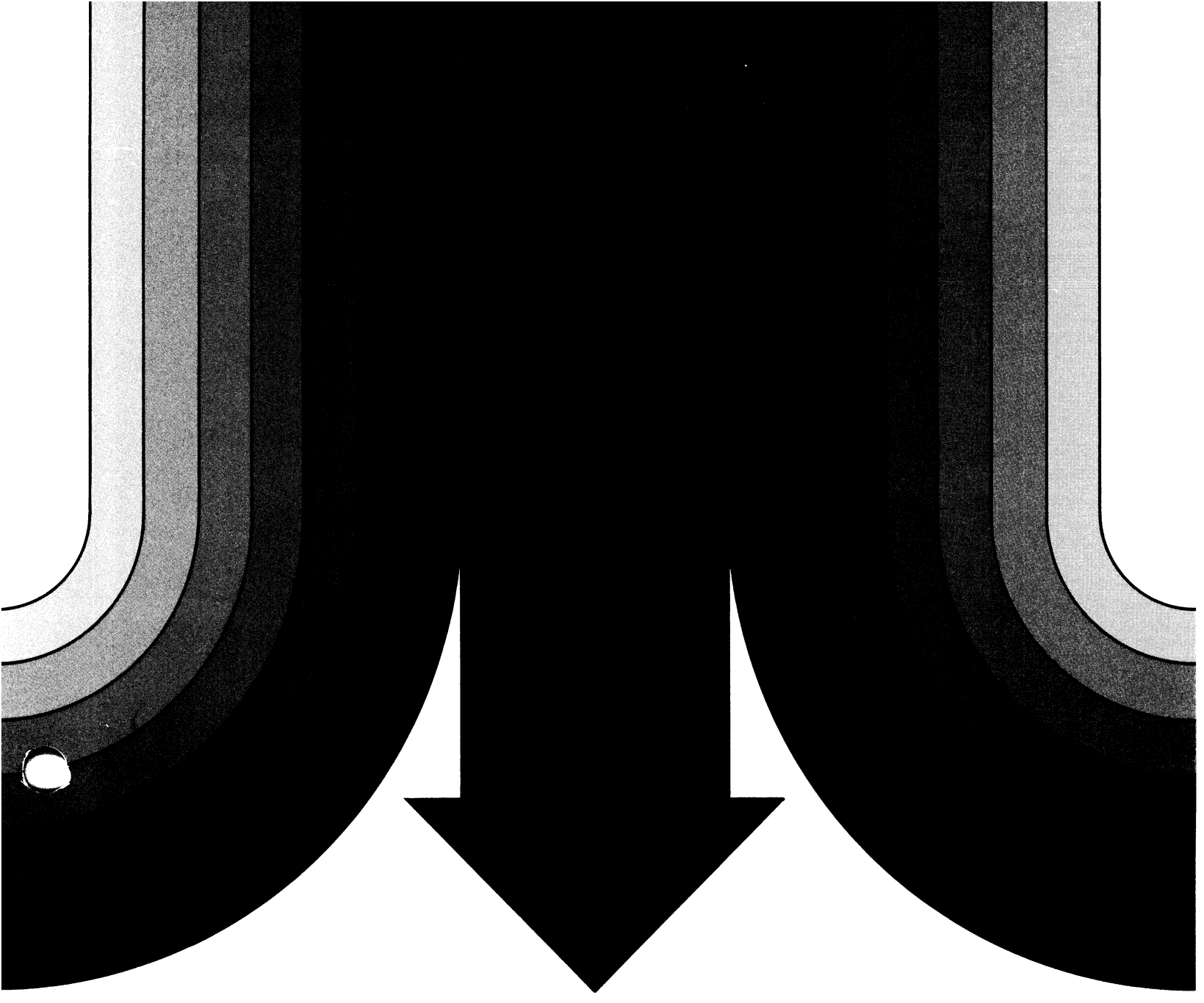


AOS/VS File System Internals

DataGeneral

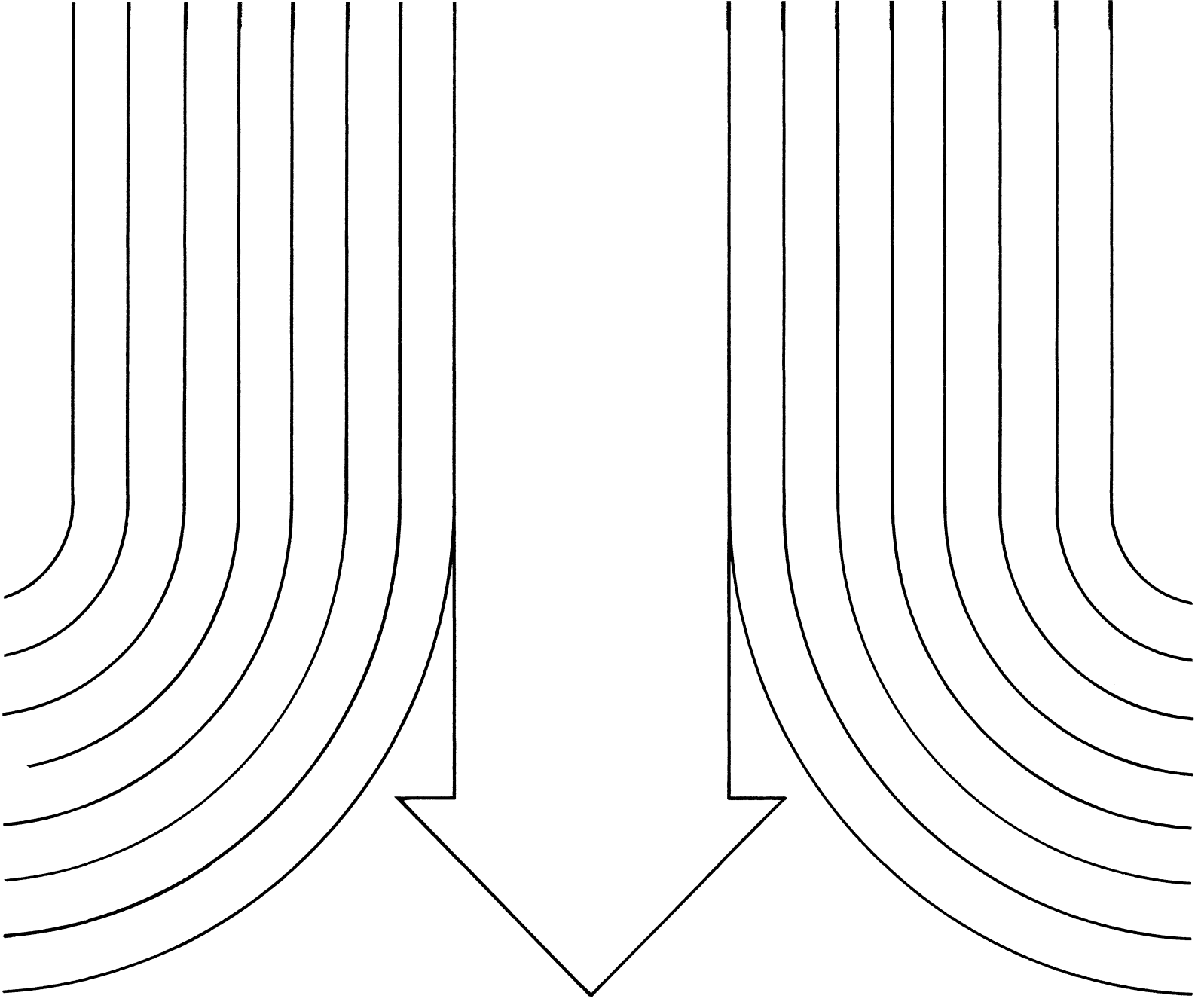
educational services

053-001002-01

NOTICE

IMPORTANT NOTICE

I UNDERSTAND THAT INFORMATION AND MATERIAL PRESENTED IN THE VS INTERNALS MANUAL MAY BE SPECIFIC TO A PARTICULAR REVISION OF THE PRODUCT. CONSEQUENTLY USER PROGRAMS OR SYSTEMS BASED ON THIS INFORMATION AND MATERIAL MAY BE REVISION-LOCKED AND MAY NOT FUNCTION PROPERLY WITH PRIOR OR FUTURE REVISIONS OF THE PRODUCT. THEREFORE DATA GENERAL MAKES NO REPRESENTATIONS AS TO THE UTILITY OF THIS INFORMATION AND MATERIAL BEYOND THE CURRENT REVISION LEVEL WHICH IS THE SUBJECT OF THIS MANUAL. ANY USE THEREOF TO YOU OR YOUR COMPANY IS AT YOUR OWN RISK. DATA GENERAL DISCLAIMS ANY LIABILITY ARISING FROM ANY SUCH SITUATIONS AND I AND MY COMPANY HOLD DATA GENERAL HARMLESS THEREFROM.



AOS/VS File System Internals

DataGeneral

educational services

053-001002-01

Table of Contents

Introduction

The AOS/VS Revision 7.50 File System constitutes a major component of the operating system. The primary function of the File System is to provide for access to and internal management of a hierarchical file structure on logical disk units. The File System is responsible for servicing all AOS/VS file and logical disk related operations, plus I/O operations on block devices (disks, magnetic tapes, MCAs, and line printers). This covers a great deal of ground due to the substantial number of AOS/VS file types and the enormous quantity of possible operations. This manual discusses in full detail the functionality and implementation of the most important File System operations.

Architecturally, the upper boundary of the File System as described in this manual includes block I/O and file/directory management, but not the device-independent or process-specific services of the AGENT. The lower boundary of the File System includes buffer and unit management, but not device drivers.

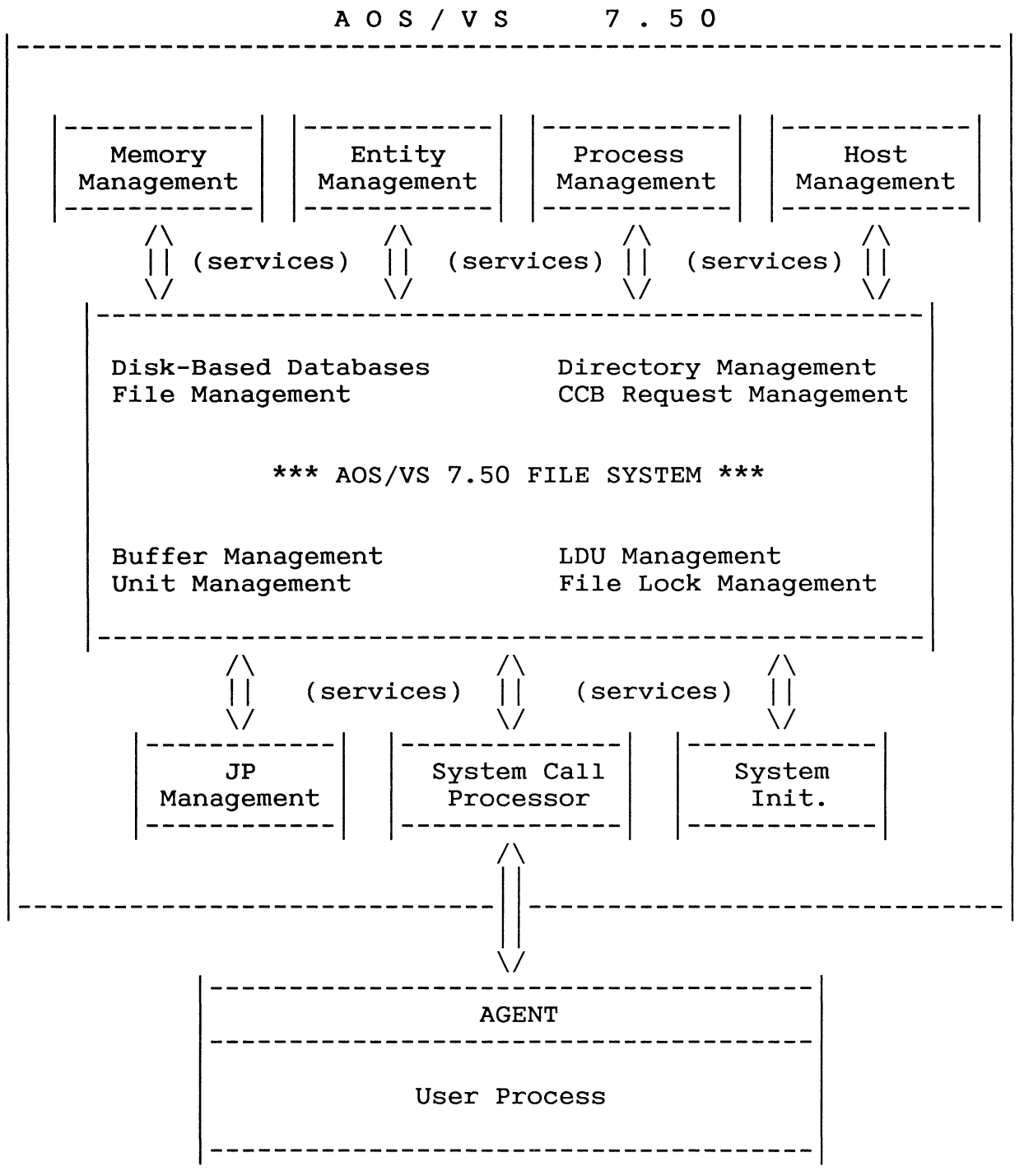
System utilities related to the File System include disk, magnetic tape, and system bootstrap programs, as well as DFMTR, FIXUP, INSTL, PCOPY, and MSCOPY. These utilities are not described in detail.

The File System makes numerous services available to external operating system components. A "service" is a subroutine which acts as an interface into the File System and which performs a specific request. For example, Process Management uses File System services to accomplish swapfile, pagefile, and IPC spoolfile I/O; System Initialization invokes File System services to initialize :BOTH, the swap and page disk; Host Management calls upon the File System to read system microcode files. The File System often invokes its own services as well. Similarly, other operating system components provide services to the File System. For example, the File System frequently invokes Memory Management services to request a chunk of system memory for database allocation, and Entity Management services to pend control blocks.

This manual presents the AOS/VS Revision 7.50 File System as a collection of individual subcomponents. The functionality, databases, services, and operations relative to each component are discussed in full detail. As a whole, presentation of the File System has generally taken a top-down approach, beginning with the high-level topics and ending with low-level topics. Each separate section, however, has been drafted in the most logical format in order to make a clear presentation. The File System subcomponents, ordered by section as they appear in this manual, are the following:

1) Disk-Based File System Databases
2) Directory Management
3) File Management
4) CCB Request Management
5) Buffer Management
6) Logical Disk Unit (LDU) Management

7) Unit Management
8) File Lock Management



1 On-Disk File System

1.1 Definitions

1.1.1 Physical Disk Unit (PU)

A physical disk unit consists of one physical disk in a single disk drive. The physical disk, as understood by AOS/VS disk drivers for the purpose of addressing, is divided into the following fundamental components: sectors, tracks, cylinders, and heads. The SECTOR on Data General disk units contains a 3-byte address header, a 512-byte data field and a 4-byte checkword. AOS/VS only recognizes the 512-byte field of the sector, which is commonly referred to as the "disk block." The other seven bytes composing the sector are controller specific. The sector is the smallest addressable unit on a disk. A TRACK consists of all the adjacent sectors on the same disk plate surface, which are of equal distance from the center of the plate. It is a circle made up of sectors. A CYLINDER consists of the set of tracks, one from each disk plate surface, which are of equal distance from the center of their respective plates. The number of cylinders per disk is equal to the number of tracks per plate. A HEAD is a small device that actually transfers bits of data to the disk. There is one head per disk plate surface on disk drives manufactured by Data General; all disk heads are colinear.

The above information is static in the Disk Information Block (DIB) of each physical unit. The AOS/VS file system only recognizes physical block addresses, not drive addresses. The disk drives must use sector, track, cylinder and head information to convert physical addresses into disk drive addresses that the controller will understand in order for effective data transfers.

DG Model 6161 Disk Storage Unit Characteristics

Bytes/Sector	512	(for all drives)
Sectors/Track	35	(varies between drives)
Cylinders	823	(varies between drives)
Heads (Tracks/Cylinder)	10	(varies between drives)

(computing from above values yields ...)

Bytes/Track	17,920	
Bytes/Cylinder	179,200	
Bytes/Disk	147,481,600	(147 Megabytes)

The disk controller references disk blocks with a sector, cylinder and head (surface) address. The first physical disk block begins at sector 0, cylinder 0, head 0. The physical layout of sectors varies for different disk models. As far as AOS/VS is concerned, sectors are logically contiguous.

1.1.2 Logical Disk Unit (LDU)

A logical disk is an association of physical disks, which creates the illusory effect of a single, contiguously addressable unit. The purpose of LDU support is to allow an extended disk address space. This concept is similar to that of virtual memory; the user is led to believe that there is unlimited space on one disk. The reality is that the AOS/VS Disk Formatter (DFMTR) constructs a string of 1 to 8 physical disks "linked together" to form one logical disk. The maximum space possible for an LDU to hold is 2**32 blocks, or 4.3 Gigablocks.

The functions of LDU management can be summarized as follows:

1) To provide a contiguous array of logical disk blocks, which span one or more physical disks and which hide disk blocks used for LDU management ("invisible space"),

2) To provide bad block remapping for logical disk blocks, which preserves the contiguous array of logical blocks in spite of isolated bad sectors,

3) To maintain the allocation status of each logical disk block and status of the LDU as a whole.

LDU management is performed not only by the AOS/VS file system and drivers, but also by indispensable system utilities:

1) DFMTR - "concatenates" physical units to create an LDU.
   Analyzes disk surface for bad blocks.
   Sets up logical disk management databases.

2) INSTL - copies disk and system bootstraps to disk.
   Copies AOS/VS default system to disk.

3) FIXUP - frees up allocated but unused disk blocks.
   Rebuilds directories after system crashes.

4) PCOPY/MSCOPY - copies an entire LDU to media of either the same or different type.

A logical disk can be grafted onto a directory with the ?INIT system call (CLI INIT command). The AOS/VS file type ?FLDU is actually a directory type file. The LDU name, specified by the user at DFMTR time, becomes the name of the directory. A subordinate directory hierarchy may already exist. Initialized LDUs are control point directories. Current and maximum space limitations are maintained like regular CPDs, except when the LDU is released, the space data is flushed to the Disk Information Block (DIB, described later) instead of the File Information Block (FIB). The master LDU (the root directory) is automatically initialized by AOS/VS during system initialization.

An "uninitialized" LDU is merely a set of physical units. In fact, it is not really a logical disk unit at all. Each unit defined in the LDU must be opened and accessed independently. When the disk unit is opened as a file, the user knows only of physical disk blocks. Physical block I/O is the only user interface. The concept of file I/O exists only on initialized LDUs.

1.1.3 LDU - PU Relationship

Physical units in an LDU may be on the same controller (DPF0 and DPF1) or on different controllers (DPF0 and DPF10). They may be of different types (DPF0 and DPJ0). Only an LDU built with a 602 MB disk cannot include other disk models. An LDU may be composed of more than one physical unit, but one physical unit may not be divided into more than one LDU (AOS/VS 7.50). There may be a minimum of one physical unit per LDU and a maximum of eight.

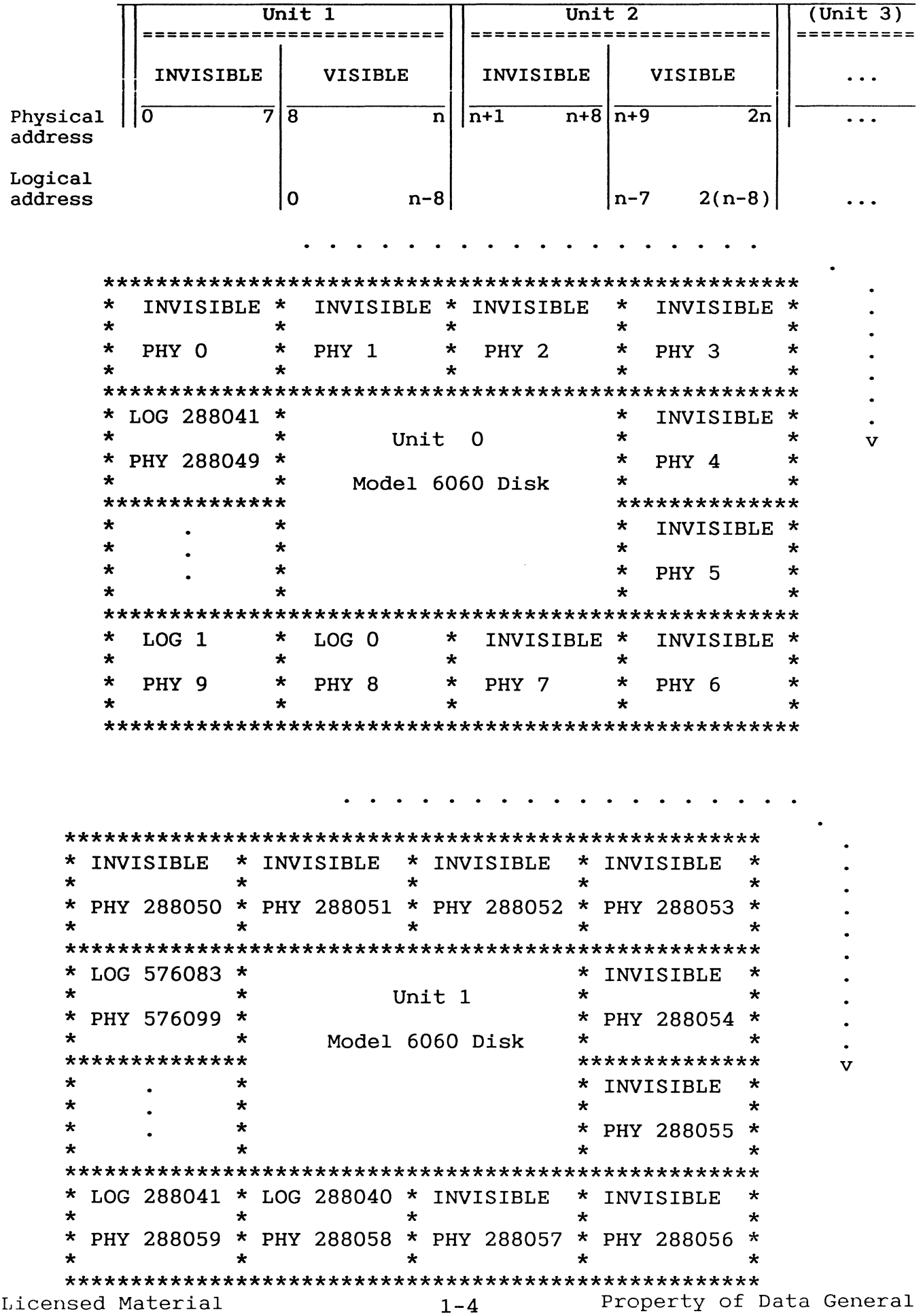
Mirrored LDUs are composed of at least two physical units. Hardware mirroring is effective only when mirroring LDU images that exist at identical physical block addresses of physical disks on the same controller. This implies that one controller contains at least two physical units, one for the primary image and one for the secondary image. Furthermore, these two physical units are mirror images of each other.

Most disk blocks of an LDU are available for use by system users; however, AOS/VS reserves the first 8 blocks exclusively for physical disk information. These blocks are called the "invisible space" since they are effectively unknown to the user. The remaining disk blocks are called the "visible space." The system also reserves several blocks in visible space for LDU specific information (see Section 1.3).

1.1.4 Logical Disk Addressing

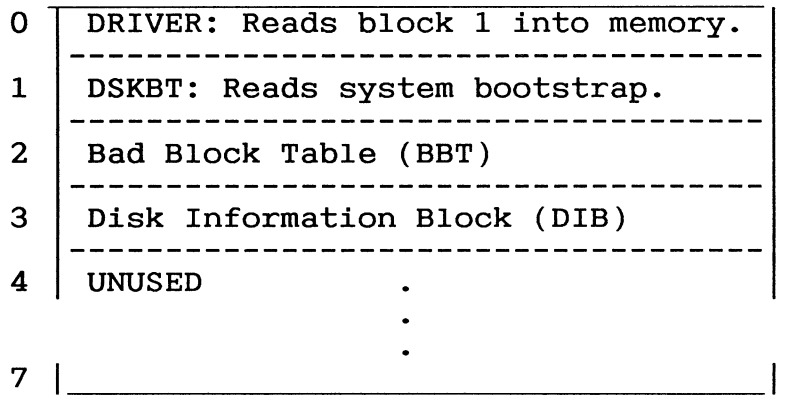
Since the first 8 physical blocks of an LDU are always reserved and cannot be accessed by the user, the user need not be aware of these blocks when accessing logical disk data. Therefore, the concept of logical disk addressing exists. Logical disk addresses, beginning with 0 and ending with n, correspond to the sequential blocks in the visible space portion of an LDU. Since the invisible space of each PU in an LDU consists of 8. blocks, logical address 0 corresponds to physical address 8. Invisible space cannot be accessed by the user on LDU type files; however, invisible disk blocks can be accessed on PUs open as separate unit files (?GOPEN @DPF20). The following example compares physical and logical addresses in an LDU (assume units of equal size). Substituting n with a simple whole number, such as 100, may help to clarify.

Multiple Unit LDU



1.2 Invisible Disk Space Structure

The invisible space of physical disk units is described as follows:



1.2.1 Disk Boot (DSKBT)

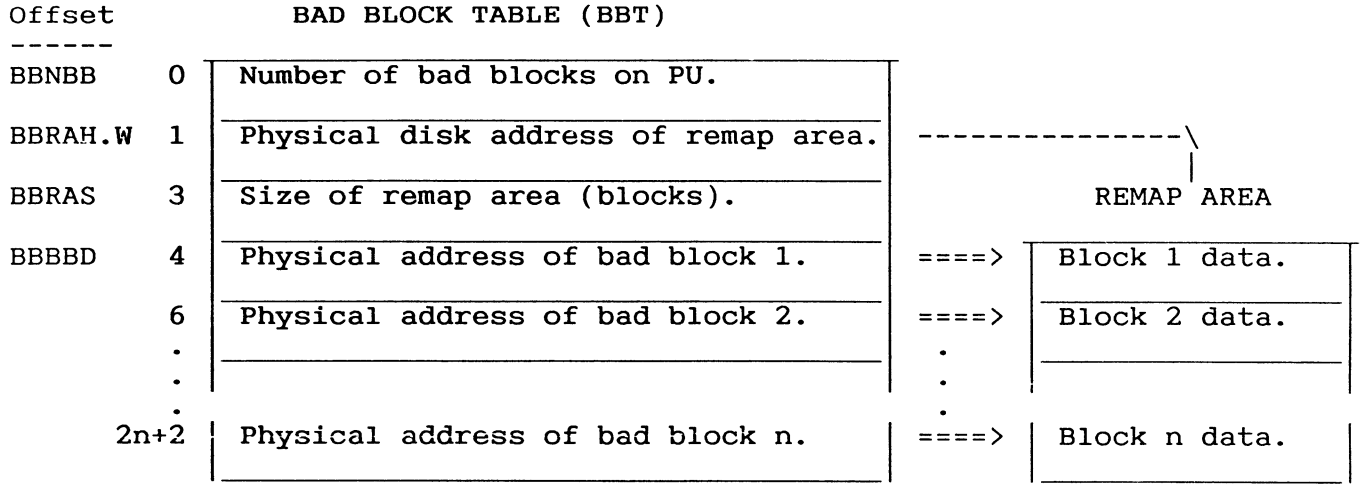
Physical block 0 consists of two parts: code to read in DSKBT (found in physical block 1) and a small disk driver. The code to read in DSKBT along with a disk-specific driver is written to block 0 by the AOS/VS Installer (INSTL). When the "BOOT device_code" command is executed from the System Control Processor (SCP-CLI), a ROM on the controller specified by the device code is activated, which loads low memory with the contents of physical block 0 and begins instruction execution at location 0377. Location 0377 contains a "JMP" instruction to the code that will read the actual DSKBT from block 1 into the next available memory location (0400). After block 1 is read into memory, flow of control is passed to the first location of block 1.

The purpose of DSKBT is to load SYSBOOT into main memory. Since DSKBT is not sophisticated enough to bring in an entire operating system, it reads and starts an intermediary program (SYSBOOT). First, DSKBT must read the DIB into main memory to obtain the starting logical disk address of SYSBOOT. Then it reads SYSBOOT into main memory (locations 0 - 075777). This is accomplished by first copying the code that will read in SYSBOOT to location 077000 and then executing it. All lower memory is then overwritten with SYSBOOT code. When SYSBOOT is completely in main memory, DSKBT simply jumps into SYSBOOT code beginning at location 2. Subsequently, SYSBOOT reads in either the installed or a user supplied AOS/VS system file and system initialization is on its way.

DSKBT, SYSBOOT and the installed AOS/VS system will only be found on physical unit 0 of an LDU.

1.2.2 Bad Block Table (BBT)

The BBT is found in physical block 2 of the physical unit. The table lists the physical addresses of bad disk blocks and the address of the contiguous remap area (in visible space). The DFMTR scans the disk for bad blocks and initializes the data in the BBT. There is a one-to-one mapping between the bad blocks listed in the BBT and the good data for that block in the remap area. Since the BBT is exactly 256. words long, the maximum number of bad blocks on any disk (except 602 MB disks) is 128. Words 0-3 of the BBT are reserved, therefore this actually reduces the number of permissible bad blocks to 126. When an LDU is initialized, the BBT of each physical unit is read in and maintained memory resident for as long as the LDU remains initialized into the system. Its memory address is found in the UDB at offset UDBBT.W. The BBT is set up as follows:



1.2.3 Disk Information Block (DIB)

The DIB is found in physical block 3 of the physical unit. The DIB contains both PU and LDU information. The primary purpose of the DIB is to provide LDU Management services with fundamental LDU information. The DIB is created and initialized by DFMTR. The most important functions of the DIB are the following:

1) Identifies the physical unit type and links it into the LDU.
2) Stores essential LDU information, such as LDU initialization status; PU sequence numbers; the logical disk address of LDU name block, ACL block and LDU Bit Map; visible/invisible space data; ADEX area data; mirroring statistics.
3) Holds LDU directory characteristics, such as the initialized directory's file type (?FLDU), hashframesize (7), data element size (1), and logical disk address of the root directory's first index block.
4) Holds system initialization and runtime information, such as the logical disk address of the system bootstrap (SYSBOOT) and the AOS/VS overlay area. It also holds a pointer to the AOS/VS installed system file.

The following diagram illustrates a compact representation of data found in DIB. (Note: Parameters to access the DIB on word boundaries are defined in PARFS.SR.)

OFFSET		DISK INFORMATION BLOCK (DIB)
IBREV	0	File system revision number.
IBTYP	1	Disk Unit Type
IBSTS	2	Status word. Always 0 and not used.
IBIDH	3	LDU Unique ID (high)
IBIDM	4	LDU Unique ID (middle)
IBIDL	5	LDU Unique ID (low)
IBSNP	6	Sequence Number of PU in LDU. Between 1 and 8.
IBNPU	7	Number of physical units in this LDU. Max of 8.
IBNHD	10	Number of heads on this PU.
IBNST	11	Number of sectors per track on this PU.
IBNCY	12	Number of cylinders on this PU.
IBVIS	13	Phys disk addr of start of visible space = 8.
IBNBH.W	14	Number of visible disk blocks on this PU.
IBBTH.W	16	Phys disk addr of Bad Block Table. Always 2.
IBUID	20	10. unused words.
IBLDF	32	LDU flags.
		NEXT 13. OFFSETS VALID ONLY ON UNIT 0 OF LDU!!!
IBNMH.W	33	Logical Disk Address (LDA) of Name Block.
IBACH.W	35	LDA of Access Control Block.
IBBAH.W	37	LDA of LDU Bit Map.
IBSBH.W	41	LDA of System Bootstrap.
IBSSB	43	Size of System bootstrap (in blocks).
IBOAH.W	44	LDA of Overlay area.
IBOAS	46	Size of overlay area (in blocks).
IBFBP	47	Pointer (IDP) to FIB of installed system.
		NEXT FCOML (13.) WORDS CONSTITUTE THE FUNNY FIB OF THE LDU. SEE SECTION 1.4.4 FOR MORE.
IBFFB	50	Start of the Funny FIB.
	51	File type. Initted to ?FLDU (013).
	52	Hash Frame Size (7).
	53	Extension for EOF in future. (Max now 32 bits)
	55	Number of bytes in file. Initted to 0.
	57	Data element size. Initted to 1.
	61	First Logical Address.
	63	Current/Maximum index levels. Initted to <0><3>.
	64	Count of inferior directories. Initted to 0.
IBCSH.W	65	Current block size of LDU. Dynamically updated.
IBMSH.W	67	Maximum block size of LDU. Initted by DFMTR.
IBDMN	71	User-defined microcode filename.
IBASZ	112	Size of ADES area (in blocks). Initted by DFMTR.
IBBMS	113	LDU Bit Map size (in blocks). Initted by DFMTR.
IBDAT.W	114	Date last mirrored.
IBTOD.W	116	Time last mirrored.
IBLDI	120	Last mirrored LDU ID.
IBMST	123	Mirror state.
IBLDL	124	Beginning of LDU list. (List of PUs in LDU)
IBMLL	164	Beginning of Mirrored LDU list. (PUs in mirror).
IBLEN	224	Length of DIB.

The valid AOS/VS disk file system revision numbers are:

SCREV (5) - Any disk with an allocated ADEX area. Support of co-resident ADEX began in AOS/VS 6.00.

SCPRV (3) - Any disk, except KISMET II, without an ADEX area.
SCKRV (4) - KISMET II type disks without an ADEX area.

If the IBREV fields of all the PUs in an LDU do not match, a partial DFMTR will abort. If IBREV contains an invalid file system revision number, LDU initialization will fail.

The disk unit type is a two-letter, ASCII representation of the disk type. For example, IBTYP contains "PJ" if the disk is an ARGUS II (i.e., DPJ type) and "PF" if the disk is a ZEBRA (i.e., DPF type).

The LDU flags, defined as bit masks, for offset IBLDF are:

IBSIN (1B0) - Logical disk initialized.
IBSBI (1B1) - SYSBOOT has been installed.
IBKS2 (1B2) - KISMET II LDU. (Special handling.)
IBDMC (1B3) - User has defined default microcode filename.
IBFXR (1B4) - FIXUP recommended on this LDU. Set at LDU release.
IBAOD (1B5) - ADEX area has been installed.
IBMIR (1B6) - Disk is part of a mirrored set of images.
IBSIP (1B7) - Mirror synchronization in progress.
IBLDU (1B8) - IBLDL and IBMLL both exist.

Two fields of the DIB, the FIB pointer to the installed system and the LDU's first file address (undefined parameter in the Funny FIB), are not DFMTR's area of concern. DFMTR initializes these locations to 0. If the LDU is to be a system disk, INSTL will take care of allocating the root directory's first index block and storing its address in the DIB. INSTL will also allocate the necessary directory data blocks and 400. extra disk blocks (default) for the installed system. INSTL will create a FIB for the installed system and place its pointer (IDP) in the root directory and in the LDU's DIB. SYSBOOT will be able to access the installed system file via the DIB. On the other hand, if the LDU is not selected to be a system disk, IBFBP will be left 0 forever, and the LDU's first address will remain 0 until the first file is created in the root directory.

Finally, the last 64. words of the DIB are a listing of the LDU and a listing of its mirror LDU (if one exists). The eight disks of the LDU will be followed by the eight disks of the mirror LDU. If no mirror exists, the second list will be filled with -1. There are four words for each disk in the LDU. The first three words are the disk type and unit number (e.g., DPF10); the fourth word is the device code. This table is used by LDU initialization code to validate the LDU configuration,

mirroring configurations, and to set up the PU-LDU relationship in in-core databases (LCB, UDB). An illustration of the concepts follows:

0	D	P
1	F	1
2	0	<0>
3		027

There are 10. more words in physical block 3, which are placed outside of the structured DIB (as defined by IBLEN), but hold information on the co-resident ADEX area. IBAIN, the ADEX installed word, is initialized to 0 by DFMTR even if there was an ADEX area allocated. The actual diagnostics are not installed now, only the area is allocated (in the last block of the disk). When the ADEX diagnostics are installed, this value will become non-zero. The ADEX reserved area in physical block 3 is defined as follows:

OFFSET		(PHYSICAL BLOCK 3)
IBADS	365	Beginning of ADEX reserved area.
IBPAH.W	374	Physical disk address of ADEX area
IBAIN	376	ADEX installed word. Initted to 0 by DFMTR.
IBAEN	377	End of ADEX reserved area.

The following illustrates the actual disk representation of the DIB. The disk type is DPM, a floppy disk, created as a single-unit LDU.

```
 00 000003 050115 000000 041514 040523 051400 000001 000001
 10 000002 000011 000050 000010 000000 001310 000000 000002
 20 000000 000000 000000 000000 000000 000000 000000 000000
 30 000000 000000 000000 000000 000176 000000 000177 000000
 40 000412 000000 000000 000000 000000 000000 000000 000000
 50 000003 000013 000007 000000 000000 000000 011000 000000
 60 000001 000000 000200 000403 000003 000000 000512 000000
 70 001107 000000 000000 000000 000000 000000 000000 000000
100 000000 000000 000000 000000 000000 000000 000000 000000
```

1.3 Visible Disk Space Structure

Several AOS/VS system databases are kept in the visible portion of the disk. These areas are allocated dynamically by either the disk formatter or the installer.

The bad block remap area, whose address is found in the BBT, is the area to which bad disk blocks are remapped. There is a remap area on each PU. The DFMTR sets up this area and hammers the address into physical block 2.

The LDU name block and the LDU access control block are only valid on the first PU of the LDU. The name specified in the name block becomes the directory name of the LDU type file when an LDU is initialized and grafted onto the AOS/VS directory hierarchy. The ACL in the access control block becomes its ACL. The DFMTR allocates these areas on the disk and stores their logical addresses in the DIB.

The bit map indicates which logical disk blocks have or have not been allocated. A set (1) bit means the block is allocated and is not free. There is one bit map per LDU and its location is set up by DFMTR. Since there are 4096 bits in a disk block, the number of blocks in the LDU bit map is calculated by applying the following equation:

```
num_bit_map_blocks = total_num_log_disk_blocks / 4096.
```

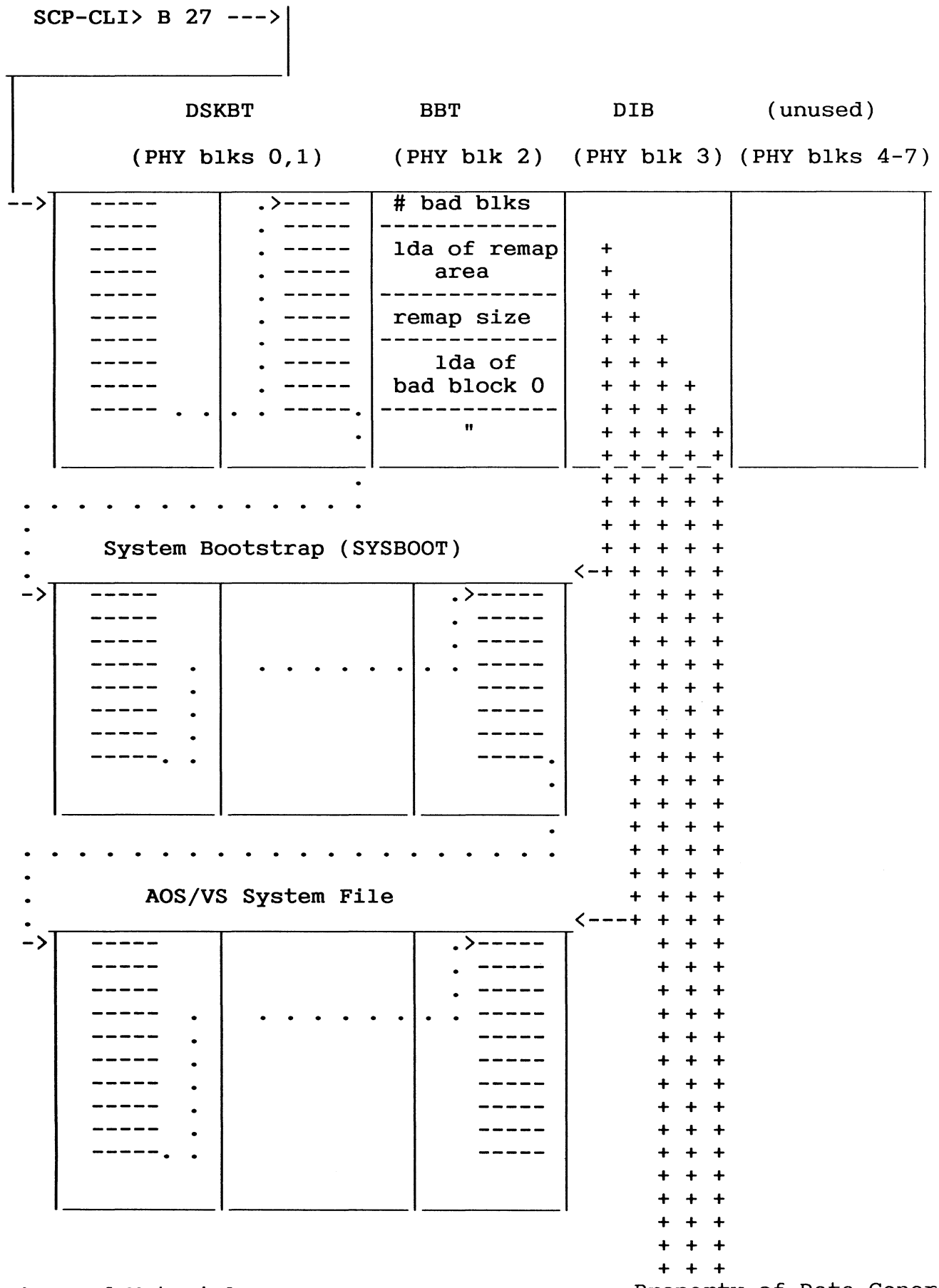
One of the DFMTR options is to designate the LDU as a system disk. If this option is selected, the DFMTR allocates 124. blocks (31K words) for SYSBOOT and stores the logical address in the DIB. INSTL will read the DIB to get this address and finally install SYSBOOT into this area. If the LDU is to be a system disk, the DFMTR also allocates space for the installed system. The default size is 400. blocks, but this can easily be changed in one DFMTR session. The logical address of this area is also stored in the DIB.

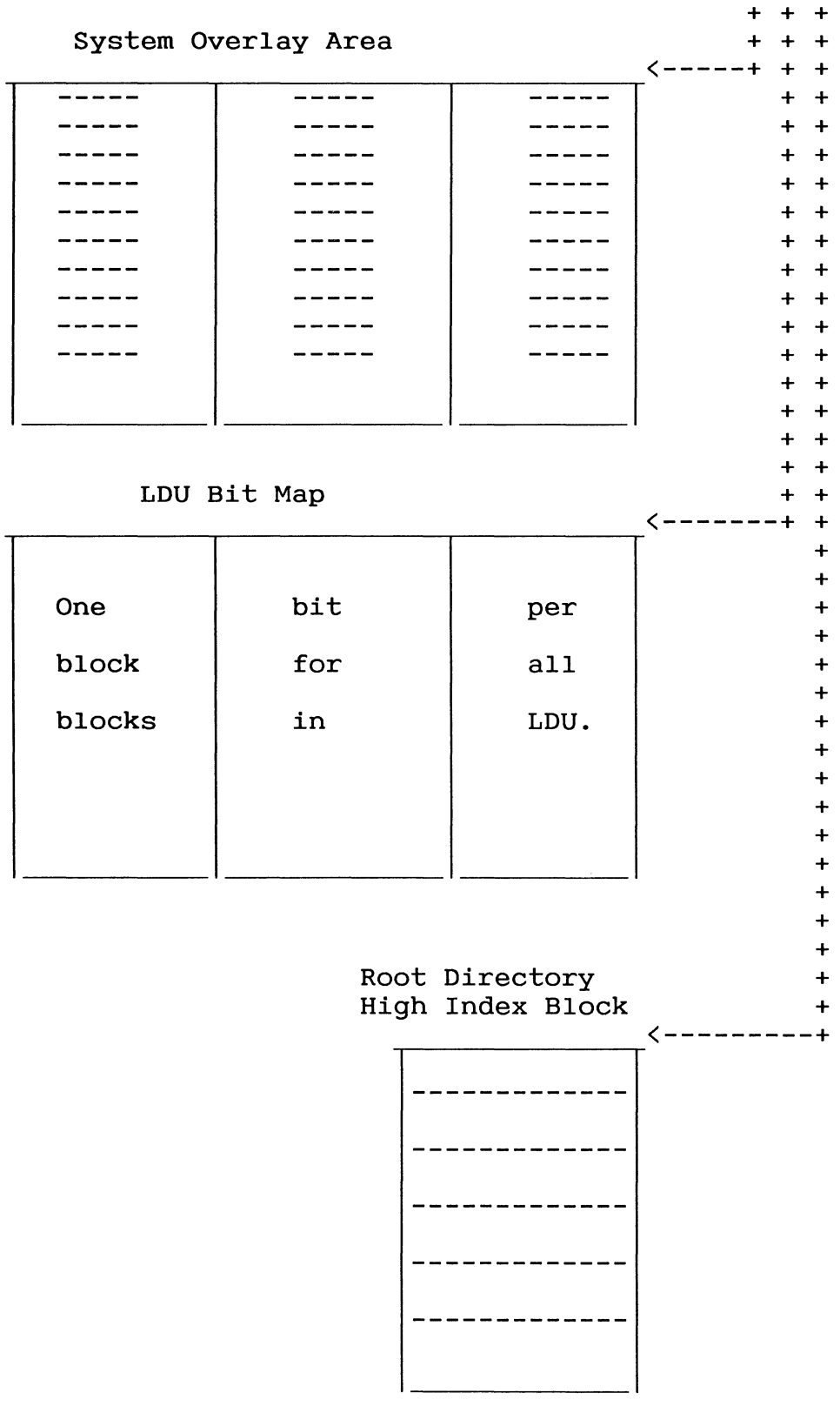
The installed system is a file that resides in the root directory (:) but has no name. When INSTL installs a system, it creates a FIB in the first general-purpose DDB of the root directory (see Section 1.4). The FIB will contain the first file address of the installed system. In order to access the FIB, INSTL must save its intra-directory pointer (IDP, see Section 1.4) in the DIB.

The co-resident ADEX area is part of neither invisible nor visible space! Locations within the ADEX area are referenced by physical address. They are not incorporated into the logical addressing scheme of the LDU. The contiguous ADEX area is allocated by the DFMTR at the end of the first physical unit. Its physical address is stored in location IBPAH.W (374) of the DIB.

The remaining blocks of visible space are free for allocation. The allocation of the first free block, which becomes the first file address (index block) of the LDU, is discussed in Section 1.2.3. The LDU is ready for general use by all AOS/VS users.

The following diagram summarizes all of the structures presented.




System Overlay Area
LDU Bit Map
One bit per block for all blocks in LDU.
Root Directory
High Index Block

1.4 AOS/VS Files

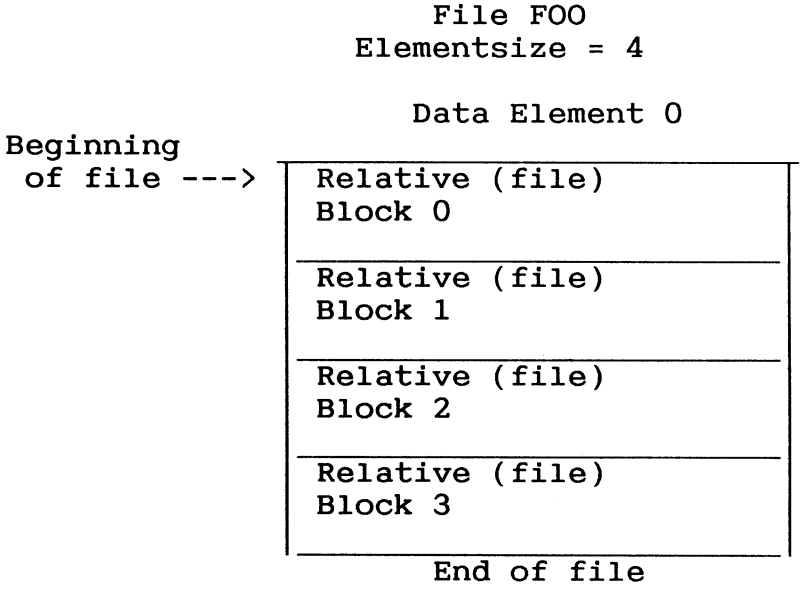
1.4.1 What is a File?

A file is a collection of logically addressable disk blocks that contain data. The n blocks of file "FOO" are logically numbered 0 - (n-1) and are not necessarily contiguous. File blocks are allocated in groups of contiguous chunks called data elements. A data element is the minimum number of contiguous disk blocks allocated or deallocated on any file I/O request. This number is represented by a file's data element size (elementsize). A file's default elementsize is a gennable system parameter. VSGEN defaults this value to 4, but accepts any valid elementsize (1 or any multiple of 4).

Data elements of the same disk file may begin at logical addresses on one physical unit of the LDU and end at a logical address on the following physical unit. Since the purpose of the LDU and logical addressing is to consolidate all the valid addresses on each physical unit, AOS/VS must support this feature.

1.4.2 File Logical Addressing

Disk files are created with a null starting address and contain no data. The first write to a file causes its first data element to be allocated, and the file's starting address becomes the logical disk address of the file's block 0. This address is stored in the File Information Block (described in Section 1.5.5). If the elementsize of file FOO is 4, then the first write to FOO will cause AOS/VS to allocate 4 contiguous disk blocks (2048. bytes), logically numbered 0-3. If FOO's elementsize were 1000., 1000. contiguous disk blocks (512000. bytes) would be allocated. VSGEN sets the default elementsize to 4.



When the first data element of a file is full, another must be allocated. However, since logical file addressing implies that contiguous allocation of all a file's disk blocks is not a requirement, a pointer to the new data element must be created. Furthermore, the AOS/VS file system must know that the second data element holds the next sequence of logically numbered disk blocks in the file. This goal is accomplished by forcing the file to undergo a structural transformation; the file grows an index level.

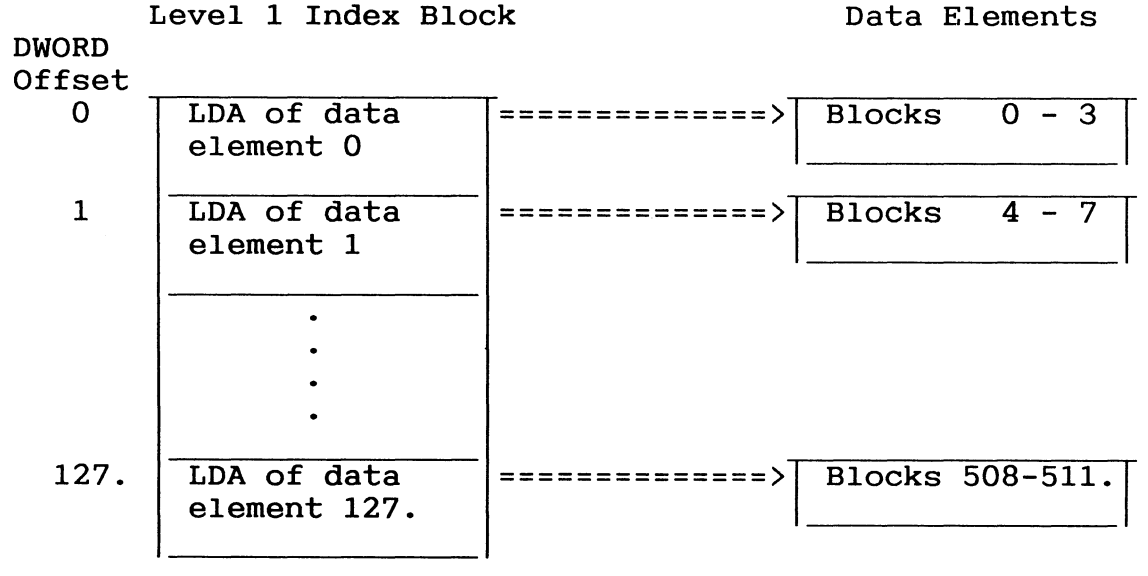
1.4.3 File Indexing

Indexing occurs when the first data element becomes filled and another must be allocated. At this stage, an index block will be allocated, and the first file address will become the logical disk address of the new index block. The index block will sequentially point to data elements 0 - n. If the first file address points to an index block, this index block is called the "high index block."

The index block is divided into 128 double-word slots, each of which contains the logical disk address of another data element. In a one-index level file, the first double word contains the logical disk address of element 0. The 128th double word contains the logical disk address element 127. If file FOO has elementsize 4, a total of 512. disk blocks (262,144 bytes) can fit in index level 1.

The following diagram illustrates one-level indexing in a file of elementsize 4.

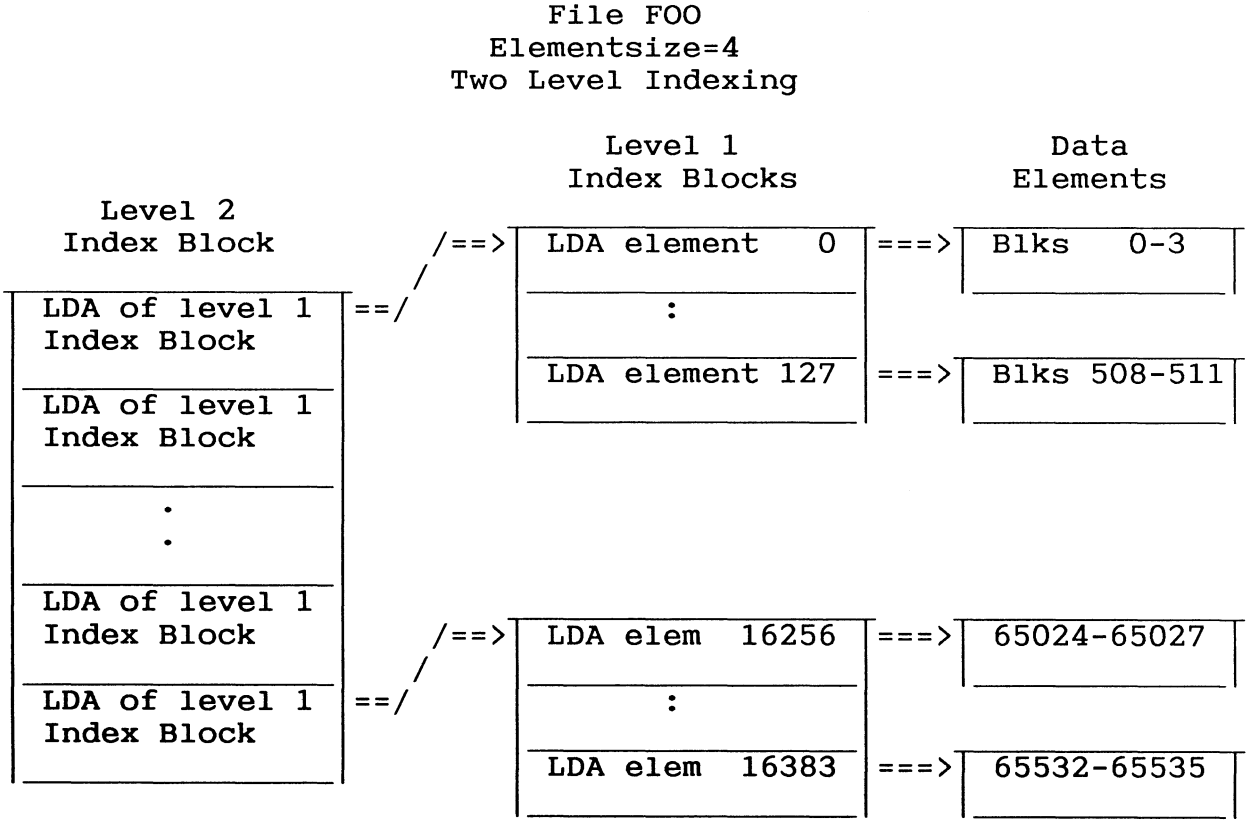
File FOO
Elementsize=4
One Level Indexing



If a file is one index level deep and grows such that the index block becomes full, the file must grow another index level. The original index block remains unaltered, but the next index block will be allocated to point to the newest data element that would not fit in the single index level format. But where is the pointer to the new index block found?

The logical disk addresses of each first-level index block are stored sequentially in yet another index block, called the second-level index block. Its logical disk address becomes the file's first address. This scheme allows for effective and efficient support of file block logical addressing.

The following diagram illustrates two-level indexing in a disk file with data element size 4. There is enough space to hold 33,554,432 bytes of data!



AOS/VS supports a minimum of 0 and a maximum of 3 file index levels. Only databases of enormous size may grow to three index levels. You can calculate that three-level indexing would allow for a maximum of 128 cubed data elements. Multiply this result by 4 to get the number of blocks, and again by 512. to get the number of bytes. This result is equal to 2**32, perfect for accommodating a file's maximum achievable byte length in the 32-bit field that AOS/VS provides. AOS/VS has reserved a field in the File Information Block for an EOF byte extension in the case of four-level indexing support.

1.4.4 Summary

The reader should now comprehend the basic structure of a file. There are many different AOS/VS file types, most of which contain independent data in the presented format. There are certain AOS/VS files, such as IPC type files, which do not require data elements, but whose essential information is maintained in its parent directory's databases. Nevertheless, one major issue remains unexplained. How does AOS/VS know where to find a file's starting logical disk address? This question can be answered by understanding the AOS/VS directory structure, presented in Section 1.5.

1.5 Directory File Structure

1.5.1 What is a Directory?

The directory file is a special AOS/VS file. The valid AOS/VS directory file types are, as defined in PARU.32.SR:

1) ?FCPD, Control Point Directory
2) ?FLDU, Logical Disk Unit
3) ?FDIR, Directory
4) ?FMTV, Mag Tape Volume

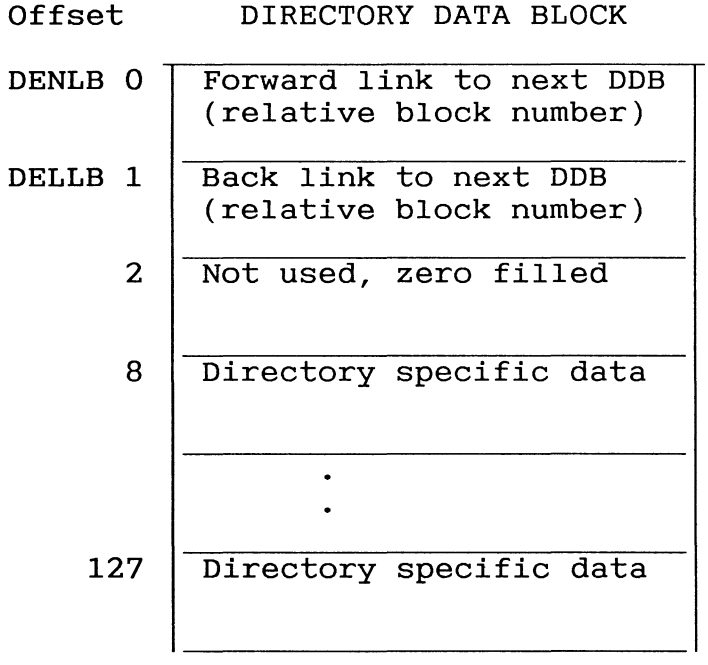
From a user's perspective, directories contain files of any AOS/VS file type, including subordinate directories. From an operating system's point of view, directories contain these files' "administrative" information. Disk data files, per se, consist solely of index blocks and actual data, while directories hold the files' characteristics (filename, ACL, link resolution pathnames, permanence, elementsize, creation time, starting address of data, etc.). All data retrieved by the CLI FILESTATUS command (?FSTAT) is maintained in the file's parent directory. Some files, such as IPC type files, do not actually contain data, but a directory entry for the file must be present in order to access it. For the AOS/VS file system to facilitate file access most efficiently, the directory file structure has some special features that differ from most other AOS/VS files. One important feature is a directory's fixed elementsize of 1, regardless of the default element size of other files. Whenever a data element is allocated to a directory file, this is the equivalent to the allocation of a single data block. Other unique characteristics will be logically presented throughout this chapter.

Only AOS/VS understands and has the ability to manipulate directory data. Directory file I/O is implicit and initiated by the operating system; other file I/O is explicit and initiated by the user.

All disk file I/O is valid only when the LDU is initialized. Otherwise, directory hierarchies and file structures are not interpreted, and only straight, physical block I/O is possible.

1.5.2 Directory Data Blocks (DDBs)

A Directory Data Block (DDB) is simply one disk block that holds directory specific data. Each individual data block in a directory can be referred to as a DDB. All DDBs begin with a standard 8 word header. The first 2 header words contain a forward and backward link to other DDBs. The rest of the DDB holds directory specific data. The following diagram outlines the general DDB structure:



1.5.3 DDB Components: Directory Data Elements (DDE)

Directory Data Elements (DDEs) are the basic building blocks of a DDB. DDEs consist of one or more contiguous nuggets: a chunk of 8 words. There are five types of DDEs, each of which contains different directory specific information. A file's DDEs maintain the essential parameters utilized by the file system for file access and control.

DDE TYPES

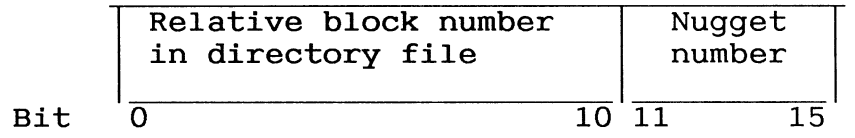
Symbol	Type code	Description	Mnemonic
DEFNB	1	File name block	FNB
DEFAC	2	File Access Control	FAC
DEFIB	3	File Information Block	FIB
DEFLB	4	File Link Block	FLB
DEFUD	5	File User Data	FUD
DEFUI	6	File Unique ID (unused)	FUI

DDEs are linked to each other by means of intra-directory pointers (IDP). Each DDE contains one or more IDPs to link other DDEs related to the same file. The 16-bit IDP is composed of two fields:

1) the relative block number in the directory file that holds the desired DDE; and

2) the nugget number in the block.

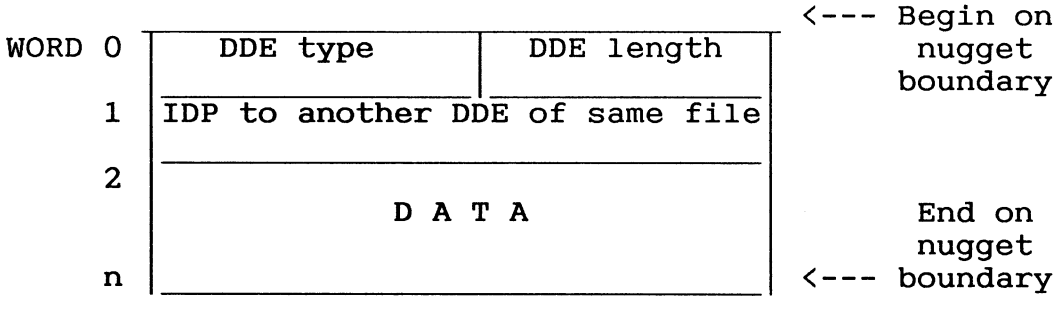
IDP FORMAT



The relative block number field is 11 bits long, imposing a maximum of 2048 possible DDBs per directory. It also prohibits directories from exceeding two levels of indexing. Although this format imposes a limit on the number of files that will fit in a directory, it still provides enough space to hold the FNBs and FIBs of over 10,000 files.

The nugget number is used to calculate the offset into the referenced DDB. This field is multiplied by 8 (nugget length) to yield the offset. Since there are 256. words in the DDB, and the first 8. words of the DDB are reserved for the header, a total of 31. nuggets fit in a DDB. The common structure of all DDEs is displayed in the following diagram:

COMMON DDE STRUCTURE



An example of an IDP would be the octal value "0345". The relative block number in the directory is 07, and the nugget number in that block is 05 (offset 050). IDPs are essentially pointers to directory data elements.

1.5.4 File Name Block (FNB)

The FNB is the DDE that holds a file's name (filename). The FNB is created and initialized at file creation and deleted at file deletion. There is one FNB per file, but many FNBs fit in one DDB. One DDB is dedicated to holding all the FNBs (and only FNBs) of files with the same hash value. The HASH VALUE of a file is calculated by taking the modulus of the sum of the ASCII values of each of its characters divided by the directory's hashframesize. Execute the following simple steps to calculate a file's hash value:

1. Sum the ASCII values of the characters in the filename.
2. Divide by hashframesize.
3. The remainder is the file's hash value.

A directory's HASHFRAMESIZE (HFS) is ideally equivalent to the number of DDBs that contain all the FNBs of the directory. For AOS/VS, the best general-purpose hashframesize is 7; hence the default directory hashframesize is 7. Hashframesize is a parameter in ?CREATE, but not in ?INIT, implying that the hashframesize of ?FDIR and ?FCPD type files can be any valid size, while that of LDUs is always 7. The maximum hashframesize is 255.

The hash value of :PMGR.PR can be calculated easily. The hashframesize of the master root LDU always defaults to 7.

1) "P"=80. "M"=77. "G"=71. "R"=82. "."=46. "P"=80. "R"=82.
   Sum is 518.

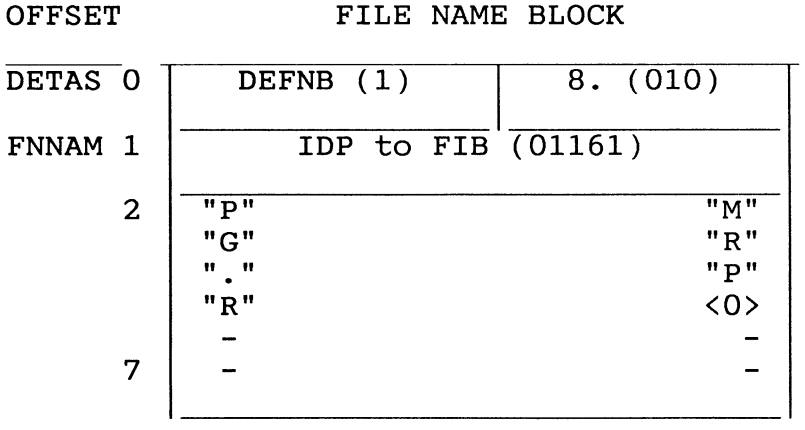
2) 518./7 = 74. remainder 0.

3) The hash value of PMGR.PR is 0.

If there are so many files with the same hash value that one DDB is not sufficient to hold them all, another DDB is allocated and linked to the original DDB, forming an FNB CHAIN. No ordering of FNB entries is attempted within a DDB; the FNB chain is searched sequentially until a filename match is encountered.

All directory and file I/O operations must read FNBs in sequential order to find the file on disk. Only two user interfaces, ?GNAME and ?GNFN, access exclusively the file's FNB and no other DDE of the file. ?RENAME must access the FIB as well as the FNB to store the pointer to the renamed file's new FNB. All other directory management interfaces access the file's FNBs, but continue on to read the FIB.

The following diagram illustrates the FNB of PMGR.PR and its actual disk representation in the DDB. Note that by breaking down the IDP (only hypothetical here), the FIB can be found in relative block 023 of the directory file at offset 0210 (021*010).



DISK REPRESENTATION OF PMGR.PR FNB

```
000 000000 000000 000000 000000 000000 000000 000000 000000
010 000410 001161 050115 043522 027120 051000 000000 000000
020 (next FNB)
```

1.5.5 File Information Block (FIB)

The FIB is the DDE that holds most of the file's "administrative" characteristics. Some of these relate to general file status information (creation time/date), while others are necessary for the I/O world's knowledge (permanence, data element size). The latter characteristics, which make up the subcomponent called the Funny FIB, are read into main memory objects (i.e., the File Control Block) when a file is opened. This information is maintained, managed, and perhaps altered during file I/O operations until the file is closed or a ?UPDATE request is made. Consequently, the FIB is flushed back to disk.

Like the FNB, the FIB is created and initialized at file creation and deleted at file deletion. When a file is not yet open, the FIB pointer (IDP) is always found in the FNB. Any file operation involving pathname resolution first must find the file entry in the directory by matching the desired filename with an FNB's contents; then, the FIB is accessed via the IDP in the FNB.

FIBs can be either 020 or 040 words long, depending on the file type. A FIB of 020 words is called a short FIB. Link files have short FIBs because all the relevant information can be found in the first 020 words; the remainder of the FIB would not be used. When a link file is accessed, AOS/VS need only obtain its resolution pathname, the parameter for which is located at offset FILBP (2).

A FIB of 040 words is called either a full-length FIB or an extended length FIB. The "normal" or full-length FIB is actually FIBLT (032) words long in AOS/VS 7.50. The extended length FIB, which contains the current and maximum lengths of control point directories, is LFIBL (036) words long. Since the length of all DDEs must fall on a nugget boundary, the resulting length for both variations is 040 words.

The following diagram illustrates the general format of the FIB and the actual disk representation of the FIB of PMGR.PR in the DDB.

OFFSET		FILE INFORMATION BLOCK (FIB)
DETAS	0	DDE type (bits 0-7) and length (bits 8-15).
FINLP	1	IDP to FNB.
FIACL	2	IDP to FAC, if not link file (or 0 if no ACL).
FILBP	2	IDP to FLB, only if link file.
FIUID	3	FIB Unique ID. Always 0.
FITCH.W	4	File Creation Time.
		Here starts the "Funny FIB."

FISTS	6	Status (Bits 0-10). Universal ACL (bits 11-15).
FITYP	7	File Type (Bits 8-15): if disk file.
FIHFS	10	Hash Frame Size: directory file.
FICPS	10	File Control Parameters: generic file index.
FICPS	10	File Control Parameters: fixed record length.
FIDCU	10	Dev Code (Bits 0-7), Unit num (Bits 8-15): unit.
FIFW1/ FIFW2	11	Extension for EOF in future.
FIEFH.W	13	Number of bytes in file (byte EOF).
FIDFH.W	15	Data element size. Set at file creation.
FIFAH.W	17	First Logical Address. Zero if null file.
FIIDX	21	Current (bits 0-7), max (bits 8-15) index levels
FIIDR	22	Count of inferior directories (dir type files).
		This is the end of the Funny FIB.

FIFUD	23	IDP to FUD (0 if no UDA).
FITAH.W	24	Time Last Accessed.
FITMH.W	26	Time Last Modified.
FIFCB.W	30	FCB address (or zero if file not open).
FICSH.W	32	Current size (blocks): control point dir file.
FIMSH.W	34	Maximum size:(blocks): control point dir file.

When a file is opened, the Funny FIB is copied into a common area in the FCB. This file system object remains resident in main memory for as long as the file is open. The Funny FIB contains the file-specific parameters that can change throughout the file's open life or that are absolutely indispensable for file I/O. These dynamic Funny FIB parameters are not modified directly in the FIB, but in the common area in the FCB. When File Management requests that the FIB be flushed (file last close, update FIB request), the Funny FIB is written back out to disk. The following status word bits are set in the FCB when the file is opened; they appear set in the FIB (offset FISTS) when flushed to disk. (Obviously, if the file is closed, some of the bits will NEVER be set!)

FBTPX (0) = Task pended on system-initiated exclusive open.
FBMDB (1) = File modified. Indicates that FIB must be flushed.
FBSXO (2) = System-initiated exclusive open.
FBPRM (5) = File is permanent. Set/reset by ?SATR.
FBDLE (6) = Delete file on last close.
FBFDB (7) = File has a UDA. Set when UDA created.
FBFLB (8) = Flush Funny FIB in FCB to FIB on disk.
This bit is set when file modified, and sometimes even if a chance it will be modified, such as on ALL write requests!
FBOEX (9) = File exclusively opened. Set by exclusive open.
FBIOP (10)= File has I/O in progress.

The last five bits of the FISTS word holds the file's universal ACL. The universal ACL holds the access control privileges allowed to all users. Since the ACL of many files, especially system files, does not incorporate any username, but consists only of a "+" template, the universal ACL feature was implemented to reduce overhead during I/O processing. Refer to Section 3.6 for more details.

The file type is also a parameter in the FIB (FITYP). This is set at file creation time and is checked on numerous occasions by AOS/VS to validate general file and I/O operations on specific file types. For example, ?GLINK is only valid for link type files; ?READ is not a valid user request for directory type files. The following pages list AOS/VS file types.

AOS/VS SYSTEM FILE TYPES

00000000000	?FLNK	LINK FILE
00000000001	?FSDF	SYSTEM DATA FILE
00000000002	?FMTF	MAG TAPE FILE
00000000003	?FGFN	GENERIC FILE NAME

DIRECTORY TYPE FILES

00000000012	?FDIR	DISK DIRECTORY
00000000013	?FLDU	LD ROOT DIRECTORY
00000000014	?FCPD	CONTROL POINT DIRECTORY
00000000015	?FMTV	MAG TAPE VOLUME - not used
00000000016	?FMDR	RESERVED FOR RT32
00000000017	?FGNR	RESERVED FOR RT32

UNIT TYPE FILES

00000000024	?FDKU	DISK UNIT
00000000025	?FMCU	MULTIPROCESSOR COMMUNICATIONS UNIT
00000000026	?FMTU	MAG TAPE UNIT
00000000027	?FLPU	DATA CHANNEL LINE PRINTER
00000000030	?FLPD	DATA CHANNEL LP2 UNIT
00000000031	?FLPE	DATA CHANNEL LINE PRINTER (LASER)
00000000032	?FPGN	RESERVED FOR RT32

IPC FILE TYPES

00000000036	?FIPC	IPC PORT ENTRY
00000000040	?FSPR	SPOOLABLE PERIPHERAL
00000000041	?FQUE	EXEC'S QUEUES
00000000042	?FGLT	LABELED TAPE
00000000043	?FGLM	LABELED MEDIA
00000000044	?FTRA	TAPE READER
00000000045	?FCRA	CARD READER
00000000052	?FTPA	TAPE PUNCH
00000000053	?FPLA	DIGITAL PLOTTER
00000000054	?FLPA	PIO LINE PRINTER
00000000055	?FLPC	LP2 LINE PRINTER(PLOTTER)
00000000061	?FCON	CONSOLE (HARDCOPY OR CRT)

NETWORK TYPE FILES

00000000063	?FREM	REMOTE HOST - REMA ACCESS
00000000064	?FHST	REMOTE HOST - X25 SVC ACCESS
00000000065	?FNPN	NETWORK PROCESS NAME
00000000066	?FPVC	REMOTE HOST - X25 PVC ACCESS
00000000074	?FSYN	SYNC LINE

DG TYPE FILES

00000000100	?FUDF	USER DATA FILE
00000000101	?FPRG	PROGRAM FILE
00000000102	?FUPF	USER PROFILE FILE
00000000103	?FSTF	SYMBOL TABLE FILE
00000000104	?FTXT	TEXT FILE
00000000105	?FLOG	SYSTEM LOG FILE (ACCOUNTING FILE)
00000000106	?FNCC	FORTRAN CARRIAGE CONTROL FILE
00000000107	?FLCC	FORTRAN CARRIAGE CONTROL FILE
00000000110	?FFCC	FORTRAN CARRIAGE CONTROL FILE
00000000111	?FOCC	FORTRAN CARRIAGE CONTROL FILE
00000000112	?FPRV	AOS/VS PROGRAM FILE
00000000113	?FWRD	WORD PROCESSING
00000000114	?FAFI	APL FILE
00000000115	?FAWS	APL WORKSPACE FILE
00000000116	?FBCI	BASIC CORE IMAGE FILE
00000000117	?FDCF	DEVICE CONFIGURATION FILE (NET)
00000000120	?FLCF	LINK CONFIGURATION FILE (NET)
00000000121	?FLUG	LOGICAL UNIT GROUP FILE (SNA)
00000000127	?FUNX	VS/UNIX FILE
00000000130	?FBBS	BUSINESS BASIC SYMBOL FILE
00000000131	?FVLF	BUSINESS BASIC VOLUME LABEL FILE
00000000132	?FDBF	BUSINESS BASIC DATA BASE FILE

CEO FILE TYPES

00000000133	?FGKM	DG GRAPHICS KERNAL METAFILE
00000000134	?FVDM	VIRTUAL DEVICE METAFILE
00000000135	?FNAP	NAPLPS STANDARD GRAPH FILE
00000000136	?FTRV	TRENDVIEW COMMAND FILE
00000000137	?FSPD	SPREADSHEET FILE
00000000140	?FQRY	PRESENT QUERY MACRO
00000000141	?FDTB	PHD DATA TABLE
00000000142	?FFMT	PHD FORMAT FILE
00000000143	?FWPT	TEXT INTERCHANGE FORMAT
00000000144	?FDIF	DATA INTERCHANGE FORMAT
00000000145	?FVIF	VOICE IMAGE FILE
00000000146	?FIMG	FACSIMILE IMAGE
00000000147	?FPRF	PRINT READY FILE

MORE FILES TYPES

00000000150	?FPIP	PIPE FILE
00000000151	?FTTX	TELETEX FILE
00000000152	?FDXF	RESERVED FOR DXA
00000000153	?FDXR	RESERVED FOR DXA
00000000154	?FCWP	CEO WORD PROCESSOR FILE

The file's data element size is a static parameter set at file creation time. It is used by CCB Request Management to determine how many blocks to read on a single request. The hashframesize (if a directory type file), the file control parameters (fixed record I/O or generic files), and device code/unit number (unit file) are static and set at file creation time as well.

There is a 32-bit parameter in the Funny FIB, which contains the byte length of a file (FIEFH.W). This field is initialized to 0 at file creation and is updated by I/O management as the file grows. The maximum file length is 2**32 bytes, or about 4.3 Gigabytes. The Funny FIB, however, contains a reserved field in the case of future four-level indexing support (FIFW1).

File I/O management checks the current/maximum index level field (FIIDX) to determine whether or not the file will grow beyond its maximum valid capacity. FIIDX is initialized at file creation to 0 current index levels and a potential maximum of 3. The count of inferior directories, FIIDR, is valid for directory type files only. This parameter is initialized to 0 at file creation and is incremented for each inferior directory created. File Deletion Services inspects this parameter before deleting a directory, because on a user request there is a restriction that a directory cannot be deleted if any of its entries are directories.

Perhaps the most important parameter in the Funny FIB is the starting file address (FIFAH.W). This field contains the logical disk address of the start of the file. If the file is null, this field will contain 0. Furthermore, unit (non-disk) files, IPC files and generic files do not care about this field, since they do not contain physical data on disk. Unit file types even take the liberty of redefining this field. If the file is not null, its first block may be either a data element or an index block. Whichever it may be, the only way the file system can access the beginning of a file's data is through this offset of the file's FIB, located in the file's parent directory!

Files that do not contain actual data on disk may need to utilize the FIB in a non-standard manner. For example, the IPC mechanism does not function through disk I/O operations. An IPC file is really just the definition of a system global port number. I/O to IPC files is initiated via ?ISEND and ?IREC system calls. The ?ILKUP system call calls upon directory management to verify the existence of the IPC file in a given directory by reading the FNB. Subsequently, it reads the FIB to retrieve the global port number. The AOS/VS file system accounts for this phenomenon by redefining some FIB parameters.

FIB PARAMETER REDEFINITIONS FOR IPC FILES

FIPHI	013	Global port number (high): IPC files
FIPLO	014	Global port number (low): IPC files

Here are some unit management redefinitions:

FIB PARAMETER REDEFINITIONS FOR MAG TAPE UNIT FILES

FILFL	013	Logical EOT File
FILBL	014	Logical EOT Block
FIFIL	015	Current File Number
FIBLK	016	Current Block Number
FIOBL	017	Old Block Count
FIOB2	020	Second Old Block Count

The following illustrates the disk representation of a possible FIB for PMGR.PR. Notice that it is located at offset 0210 of the DDB, as its IDP in its FNB (previous section) indicated.

DISK REPRESENTATION OF PMGR.PR FIB

```
000 000022 000007 000000 000000 000000 000000 000000 000000
***
210 001440 000005 001165 000000 010465 063137 000003 000101
220 000000 000000 000000 000000 104000 000000 000004 000000
230 006274 000403 000000 000000 010664 062776 010664 074612
240 000000 000000 000000 000000 000000 000000 000000 000000
250 (next DDE)
```

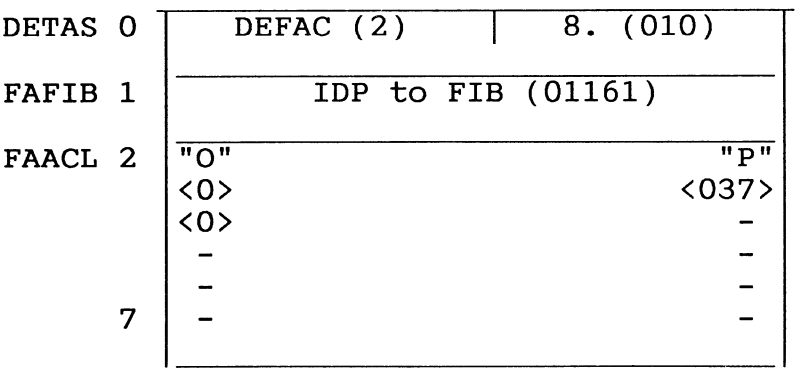
1.5.6 File Access Control Block (FAC)

The FAC, whose IDP is found in the FIB, is the DDE that holds a file's non-universal ACL. For example, if the file's ACL were "USERNAME,OWARE +,RE", "USERNAME,OWARE" would be stored in the FAC, but "+,RE" would be stored in offset FISTS of the FIB. If the ACL were only "+,RE", no FAC would be allocated. Since access privileges cannot efficiently be represented by the ASCII values of "OWARE", five standard bit positions have been defined:

```
APOWN (013) = Owner   access
APWRT (014) = Write   access
APAPN (015) = Append  access
APRED (016) = Read    access
APEXC (017) = Execute access
```

The direct user interface to a file's FAC is through the ?GACL and ?SACL system calls, which get and set the file's ACL, respectively. Whenever access privilege checking is done, an FAC must be accessed (if it exists).

The following diagram illustrates a possible FAC of PMGR.PR and its actual disk representation in the DDB. Inspection of PMGR's FIB will allow the complete ACL to be established: "OP,OWARE +,RE." Refer to Section 3.6 for more details.



DISK REPRESENTATION OF PMGR.PR FAC

```
000 000000 000000 000000 000000 000000 000000 000000 000000
***
250 001010 001161 047520 000037 000000 000000 000000 000000
260 (next DDE)
```

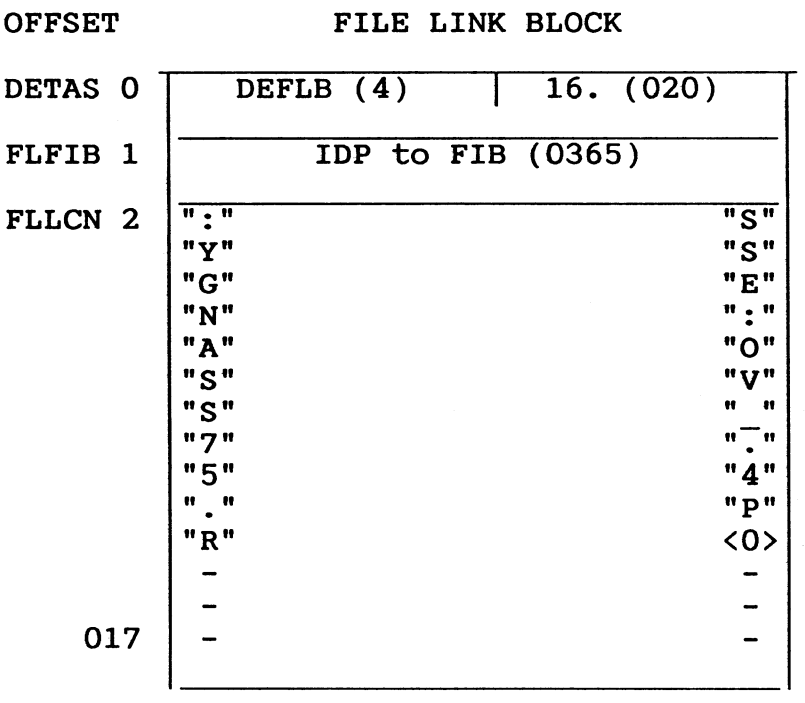
1.5.7 File Link Block (FLB)

The FLB is the DDE that contains the resolution pathname of a link type file. The FLB is accessed through an IDP stored in the file's FIB (offset FILBP). The FLB is created at file creation and deleted at file deletion.

Link files have no ACL; AOS/VS translates the link filename into its resolution pathname, and the ACL of the latter is checked against the caller's access privileges. Therefore, the FLB IDP in the FIB (FILBP) is defined at the same offset as the FAC IDP (FIACL). If the file being accessed is a link file, offset 2 of the FIB contains the IDP to its FLB; if the file is of any other type, offset 2 contains the IDP to its FAC.

The direct user interface to a link file's FLB is through the ?GLINK system call, which retrieves its contents. Virtually all other system and user interfaces with FLBs occur whenever a link file is encountered during pathname resolution.

The following diagram illustrates the FLB of a file with a resolution pathname :SYSGEN:AOSVS_7.54.PR and an actual disk representation in the DDB.



DISK REPRESENTATION OF FIB AND FLB IN DDB

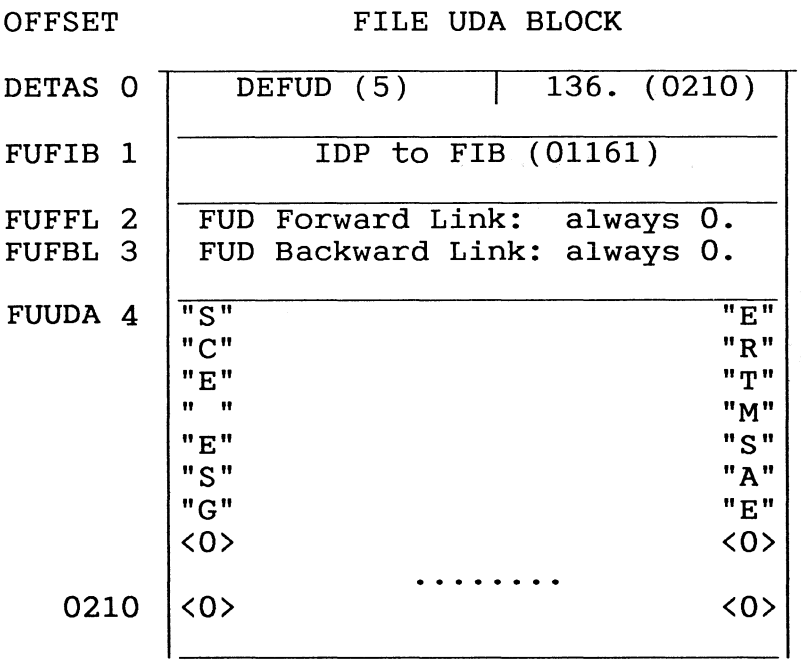
```
DDB: 000 000000 000000 000000 000000 000000 000000 000000 000000
FIB: 010 001420 001001 000343 000000 010465 063227 000003 000112
     020 000000 000000 000000 000005 034000 000000 000004 000000
FLB: 030 002020 000341 035123 054523 043505 047072 040517 051526
     040 051537 033456 032464 027120 051000 000000 000000 000000
     050 (next DDE)
```

1.5.8 File UDA Block (FUD)

The FUD is the DDE that holds a file's user data area (UDA). UDAs are exactly 128. words long and can consist of any binary data. Any file except link files can have a UDA. The FUD IDP is found in the FIB (offset FIFUD). Since UDAs take up more than one half of a disk block of space, only one FUD can fit in a DDB. AOS/VS allocates a new DDB whenever a new UDA is to be created. However, a FUD may not be the only DDE in the DDB.

A UDA is created via the ?CRUDA system call interface. The UDA can be read by issuing the ?RDUDA system call and written by issuing ?WRUDA. Once a UDA has been created, it cannot be deleted until the entire file is deleted.

Suppose you created a UDA for :PMGR.PR that said, "SECRET MESSAGE." The following diagram illustrates how it would look.



DISK REPRESENTATION OF PMGR.PR FUD

```
000 000000 000000 000000 000000 000000 000000 000000 000000
010 002610 001161 000000 000000 051505 041522 042524 020115
020 042523 051501 043505 000000 000000 000000 000000 000000
030 000000 000000 000000 000000 000000 000000 000000 000000
***
220 000000 000000 000000 000000 000000 000000 000000 000000
230 (next DDE)
```

1.5.9 Directory Bit Map

One block in every directory is dedicated to holding the Directory Bit Map. The first 2048. bits of the bit map block (128. words) monitor the size of the directory by maintaining a record of which DDBs are allocated and which are free. A one bit means that the relative block number corresponding to that bit position has been allocated. When a new DDB must be allocated from disk, the directory bit map is searched for a zero bit. If a zero bit is found, it is set and the block is allocated to the directory as a DDB. It can now be filled with DDEs.

DDBs are allocated neither randomly nor sequentially. DDBs for FNBs are allocated in every fourth word of the bit map, starting at word 0. This means that FNB DDBs are allocated as relative blocks 0-15, 64- 79, 128-143, etc. However, the first HFS blocks (0 through HFS-1) of the directory are always reserved for FNB DDBs, even if hashframesize is 1! Each of the FNBs created in the first HFS-1 blocks are called ROOT FNBs because the DDB in which they are contained is the first FNB DDB for its hash value. If a file is created whose FNB overflows the root

FNB DDB, another DDB is allocated (still exclusively for FNBs of the same hash value). The "overflow" FNBs are called NON-ROOT FNBs.

The first general-purpose DDB (used for FIBs, FACs, FLBs and FUDs) always immediately follows the last root FNB DDB. It is always allocated in relative block HFS. For example, if HFS is equal to 7, the first general-purpose DDB is allocated in relative block 7. This block, called the FIB ROOT, marks the beginning of the FIB CHAIN. All remaining general-purpose DDBs allocated in this directory will be linked to the FIB root.

The directory bit map always corresponds to relative block number HFS+1, the DDB following the FIB root. The bit map block and the FIB root block are exceptions to the rule that FNB DDBs are reserved for relative block numbers 0-15. If HFS is greater than 15., the rule is not broken.

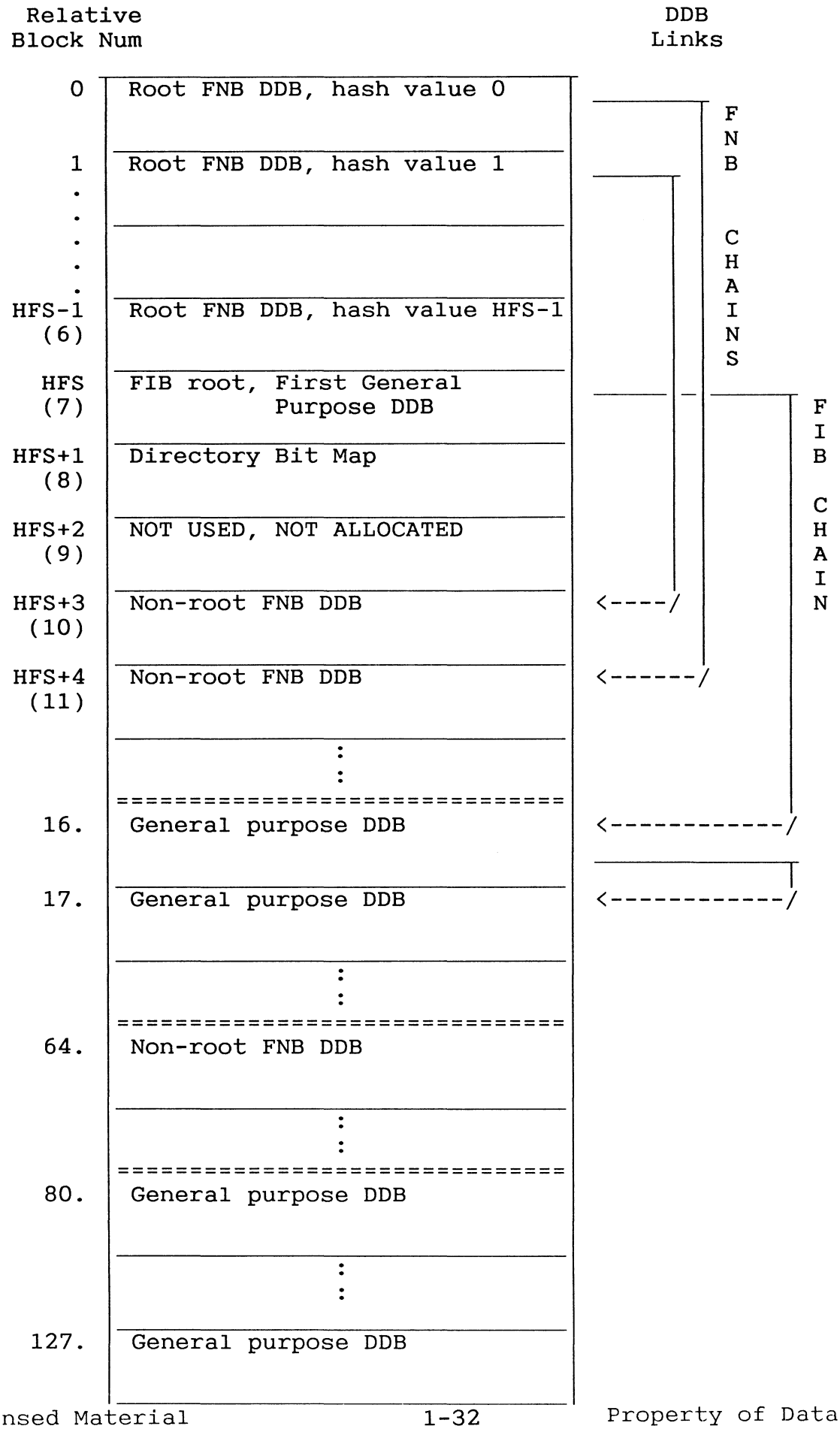
1.5.10 AOS/VS Directory Structure: The Global Picture

The previous sections have described the basic concepts and building blocks of a directory in great detail. This section will tie all these individual pieces together to present the complete, structured format of the AOS/VS directory file.

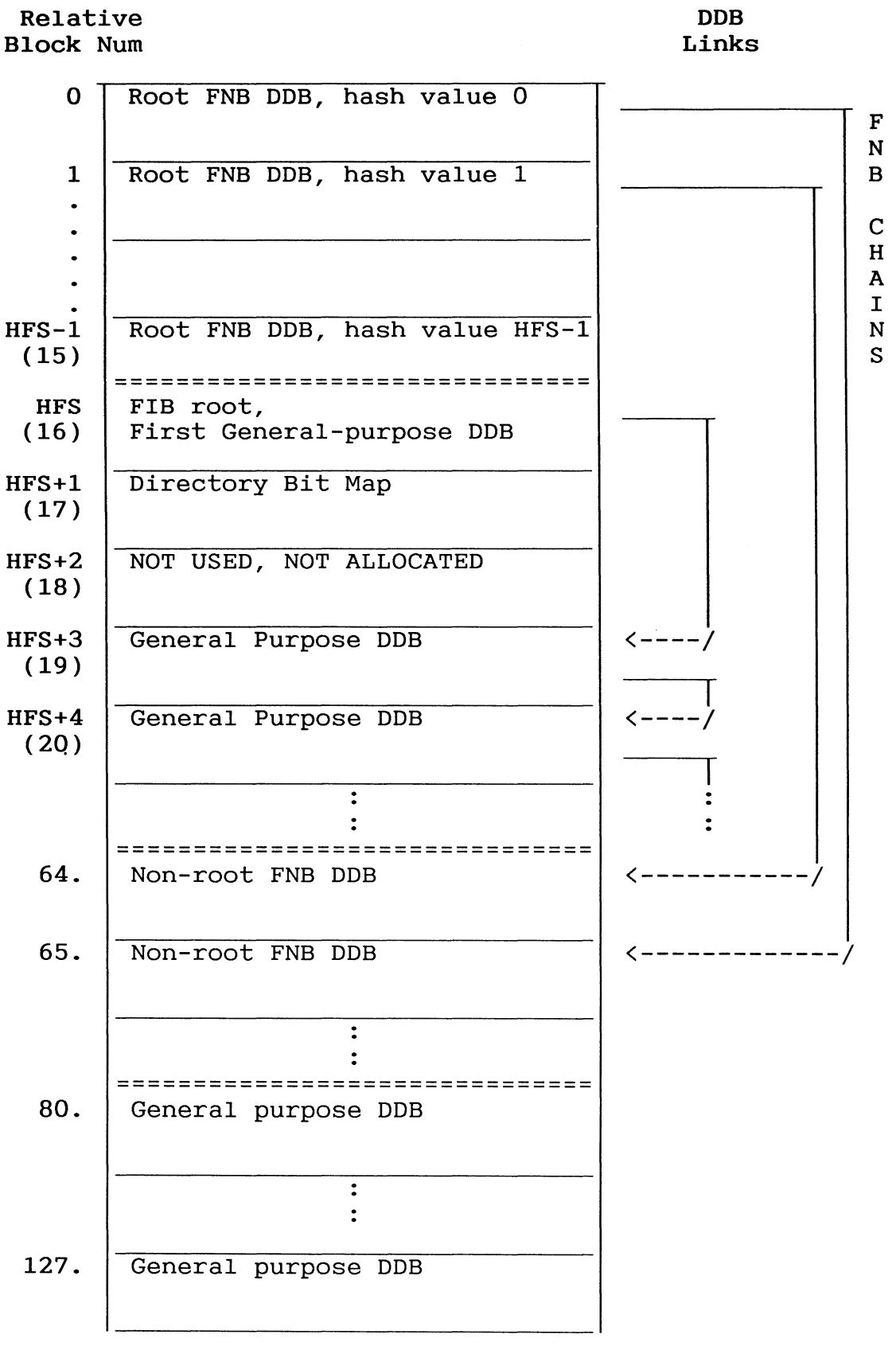
Directories are always at least one index level deep. Due to the intricate structure of directory files, the DDB allocation mechanism, and the standard elementsize of 1, it is impossible for a directory to exist with less than one level of indexing. Files are created with no data and a null starting address; this is true for directory files as well. However, upon the first file creation, the minimum number of directory blocks created is 3: one for the FNB DDB, one for the FIB DDB, one for the directory bit map. Considering the elementsize of 1, one level indexing will naturally take shape. It is important to remember that the 16-bit format of the IDP limits the directory's size to 2048. blocks and two levels of indexing.

The layout of the DDBs in a directory can best be understood by illustration. The following diagrams display the format of a one-index level directory. Two diagrams are presented: one of a directory with hashframesize 7 to exhibit DDB utilization within the first 16 relative blocks, and the other of a directory with hashframesize 16. The DDBs separated with double lines fall on bit map word boundaries.

AOS/VS Directory File Structure: Hashframesize = 7

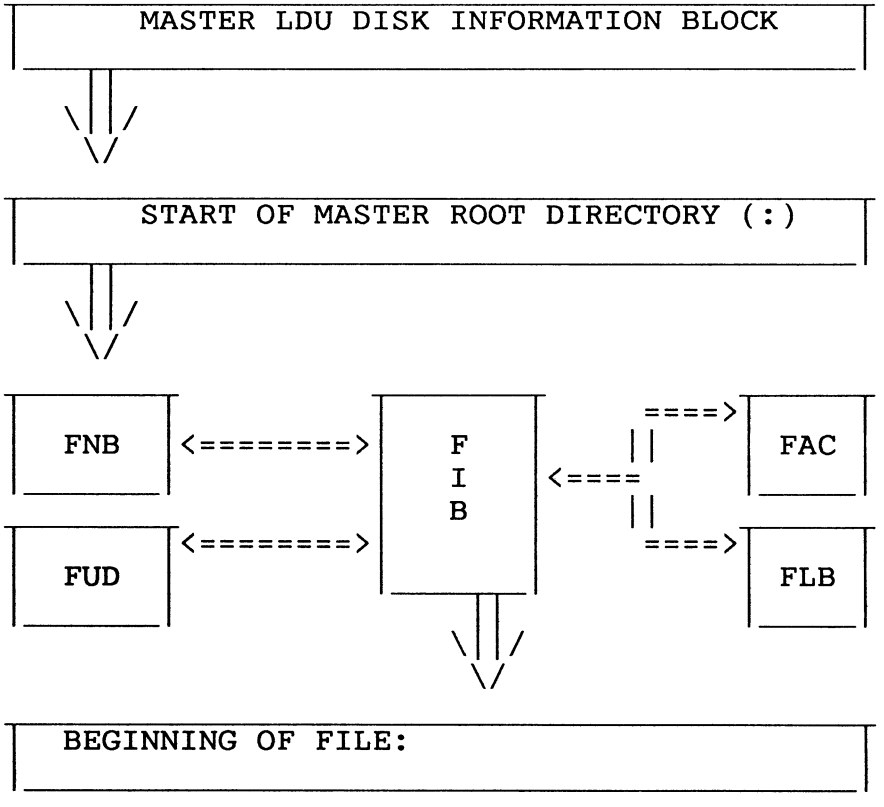


AOS/VS Directory File Structure: Hashframesize = 16.



Remember that the word 0 of the directory bit map is reserved for FNB DDBs. The root FIB DDB is always located in relative block number HFS. However, the relative block number of the next general-purpose DDB depends on the directory's hashframesize.

Finally, the following picture summarizes the connections between DDEs of all types, as well as the AOS/VS directory structure as a whole. Beginning with the DIB (physical block 3) of the logical disk unit, where the starting address of the root directory is located, the illustration shows the logical traversal of the directory hierarchy.



1.6 Locating File Contents

Logical and physical disk addressing, directory structures, file structures, and indexing have been presented in previous sections. You should now understand the AOS/VS physical disk structure enough to be able to traverse the directory hierarchy and locate any disk file beginning with the root directory. The purpose of this section is to clarify any confusion in this regard, by demonstrating how traversing the directory hierarchy is done. Beginning with the physical block 3 (the DIB), the systematic procedure of locating the contents of a disk file on the logical disk will be explained. The final diagram will mark each step in the procedure to render "the big picture." For the sake of simplicity, all directories in this example are only one index level deep.

LOCATING FILE :UTIL:XHELP.CLI

(1) Find the root directory's first address.
Remember that the DIB contains a series of common offsets within a FIB, the Funny FIB. This area holds essential status information for the LDU and is only valid on the first PU of the LDU. Offset 61 of the DIB (IBFFB + 011) contains the logical address of the LDU directory's high index block. If the LDU is the master root (:), offset 61 points to the system root directory.

(2) Find the FNB in which UTIL resides.
Since the hash value of "UTIL" is 3, "UTIL" is found in relative block 3 of the root directory (if it exists, of course).

(3) Scan the FNB for "UTIL."
If it is not found, check the forward link word in the DDB for the next DDB of the FNB chain. If it is 0, then :UTIL does not exist. Otherwise, scan the next DDB in the FNB chain until either the filenames are exhausted or "UTIL" is found. Once it is found, retrieve and break down the IDP to its FIB.

(4) Find the FIB for "UTIL."
The IDP to the FIB is 0345. This breaks down to the FIB's relative block number (7) in the first 11 bits and its nugget offset (5) in the DDB. Since the relative block number is 7, examine offset 14. (016) of the root's index block, which holds the logical disk address of the desired DDB. Then, since 5 * 8 (nugget size) = 40. (050), the beginning of the FIB is found at offset 050 of the DDB.

(5) Find the starting logical address of :UTIL and go to it.
This address is located at offset FIFAH.W of the FIB.

(6) Find the FNB for "XHELP.CLI."
Its hash value is 3.

(7) Scan the FNB for "XHELP.CLI."

(8) Find the FIB for "XHELP.CLI."
The IDP breaks down to relative block number 026 and DDB offset 0110.

(9) Find the starting logical address of :UTIL:XHELP.CLI.
Examine its contents!

Master Root LDU's DIB: Physical Block 3

```
   00 000003 050115 000000 041514 040523 051400 000001 000001
   10 000002 000011 000050 000010 000000 001310 000000 000002
   20 000000 000000 000000 000000 000000 000000 000000 000000
   30 000000 000000 000000 000000 000176 000000 000177 000000
   40 000412 000000 000000 000000 000000 000000 000000 000000
   50 000003 000013 000007 000000 000000 000000 011000 000000
/= 60 000001 000000 000200 000403 000003 000000 000512 000000
|  70 001107 000000 000000 000000 000000 000000 000000 000000
| 100 000000 000000 000000 000000 000000 000000 000000 000000
| ***
|
\=======>     (1) Start of root directory: LOG BLK 200

/=  00 000000 000360 000000 000357 000000 000000 000000 000201
|   10 000000 000000 000000 000000 000000 000000 000000 000203
|   20 000000 000202 000000 000000 000000 000000 000000 000000
|   ***
|
\=======>     (2) DDB of FNBs with Hash Value 3: LOG BLK 0201

    00 000000 000000 000000 000000 000000 000000 000000 000000
    **          (3)
/=  40 000410 000345 052524 044514 000000 000000 000000 000000
|                    "U""T" "I""L" <0><0> <0><0> <0><0> <0><0>
|   50 000410 001115 044105 046120 000000 000000 000000 000000
|                    "H""E" "L""P" <0><0> <0><0> <0><0> <0><0>
|   60 000410 001121 044517 050115 043522 027123 052000 000000
|                    "I""O" "P""M" "G""R" ".""P" "R"<0> <0><0>
|   ***
|
\=======>
```

(4) Root Dir Relative Block 7: LOG BLK 0203

```
 00 000000 000000 000000 000000 000000 000000 000000 000000
***
 50 001440 000005 000351 000000 011763 043624 000003 000014
 60 000000 000000 000000 000000 031000 000000 000001 000000
 70 000255 000403 000005 000000 013617 043427 013617 043427
100 000000 000000 000000 000153 000002 044760 000000 000000
110 001010 000345 047520 000037 000000 000000 000000 000000
***
```

\=======> (5) Start of :UTIL: LOG BLK 0255

```
 00 000000 000235 000000 000343 000000 000272 000000 000527
 10 000000 000727 000000 000463 000000 000350 000000 000240
 20 000000 000237 000000 000000 000000 000000 000000 000000
 30 000000 000000 000000 000000 000000 000000 000000 000000
 40 000000 000262 000000 000267 000000 000460 000000 000267
 50 000000 000244 000000 000466 000000 000400 000000 000623
 60 000000 000357 000000 000464 000000 000434 000000 000410
 70 000000 000635 000000 000636 000000 000242 000000 000000
100 000000 000000 000000 000000 000000 000000 000000 000000
***
```

\=======> (6) DDBs of FNBs with Hash Value 3: LOG BLK 0527

```
 00 000000 000000 000000 000000 000000 000000 000000 000000
***
140 000000 001305 050123 054523 042522 046505 051456 047502
                  "P""S" "Y""S" "E""R" "M""E" "S""." "O""B"
150 000000 001331 051503 047515 027120 051000 000000 000000
             (7)  "S""C" "O""M" ."".""P" "R"<0> <0><0> <0><0>
160 000410 001311 054110 042514 050056 041514 044400 000000
                  "X""H" "E""L" "P""." "C""L" "I"<0> <0><0>
170 000000 001345 043103 052456 050122 000000 000000 000000
                  "F""C" "U""." "P""R" <0><0> <0><0> <0><0>
***
```

\=======>

(8) :UTIL Dir Relative Block 026: LOG BLK 0400

```
  00 000000 000000 000000 000000 000000 000000 000000 000000
 ***
/= 110 001440 000005 000351 000000 011763 043624 000003 000014
 | 120 000000 000000 000000 000000 031000 000000 000001 000000
 | 130 000672 000403 000005 000000 013617 043427 013617 043427
 | 140 000000 000000 000000 000153 000002 044760 000000 000000
 | 150 001010 000345 047520 000037 000000 000000 000000 000000
 | ***
 |
 |
 |
\=======>     (9) Data Element 0 of :UTIL:XHELP.CLI, LOG BLK 0672

  00 055441 062561 072541 066054 022460 027445 026135 005040
  10 020133 020545 070565 060554 026054 022461 022535 005040
  20 020040 020133 020545 070565 060554 026054 061557 066555
  30 062556 072135 020130 044105 046120 020167 064564 064157
  40 072564 020141 071147 072555 062556 072163 005011 004440
  50 020040 020040 020154 064563 072040 067546 020145 074145
  60 061440 064145 066160 020164 067560 064543 071412 020040
  70 020040 055441 062556 062135 005012 020040 064145 066160
 100 020052 042530 042503 005040 020133 020545 066163 062535
 110 005040 020040 020133 020545 070565 060554 026054 061557
 ***
 350 061557 066555 062556 072040 043151 071163 072040 071545
 360 062440 064546 020164 064145 020141 071147 020151 071440
 370 072556 065556 067567 067054 020157 071040 060440 067157
 400 067055 072556 064561 072545 020141 061142 071145 073151
 ***
1220 022460 056045 020045 030445 005133 020545 067144 056412
```

2 Directory File Management

2.1 Overview

The AOS/VS directory file is physically structured like any other data file. Directory file I/O is performed by requesting the same general file system services that operate on any disk data file. The conceptual design of a directory file, however, differs from that of other files in its assumption of the contents of specific data blocks and its strictly observed ordering of these blocks. The previous chapter introduced the particular information maintained and manipulated by directory files as well as a blueprint of the internal disk structure of a directory. This chapter will present the internals of directory file management in relation to its interface with other subcomponents of the file system, other large components of the operating system, and the user process.

Directory file management of directory data blocks and elements relies upon allocation and deallocation operations, provided at the lowest level of directory file management, to build the file's internal structure with consistent integrity. External modules call upon these operations not only to access, but to create, delete and modify the contents of the directory as well. The calling modules are responsible for filling in the space with the appropriate directory data (DDEs). Finally, these calling modules actually constitute file system services, which are accessed either implicitly by a user process (via a system call) or explicitly by some external system path.

The user interface to any operating system service is, by definition, a system call. When a user creates a UDA via the ?CRUDA system call, the system invoked service, CRUDA.P, allocates a new directory data block, links the DDB into the FIB chain, modifies the directory bit map to reflect the new DDB, and creates a DDE of type FUD. UDAs form part of directories and "UDA I/O" is a subcomponent of directory management.

The system interface to directory management services is very straightforward. Any AOS/VS component desiring a directory management service simply sets up some accumulators and makes a subroutine call. For example, while a user-initiated request will pass through the AGENT, run through system call processing code, and ultimately jump through the system call dispatch table (MCCT.W), the system directly requests service via an "LJSR service" instruction. The latter interface usually requires more knowledge of the ramifications surrounding the request, such as database locking or buffer header management implications. Although the effect of the service is the same, the system's entry point always differs from the user's (system call) entry point.

All AOS/VS components use file system services. "File" implies "disk," and how can an operating system exist without a disk? File system services, and all operating system services in general, have been designed with easy, clean interface mechanisms for universal access.

2.2 Directory Management Databases

Directory Management services are directly related to the on-disk databases as they create, delete and modify them as the particular service prescribes. Nevertheless, there are various other databases more closely associated with the file I/O subcomponent of data management. These databases are accessed and utilized by directory management services when the file on which a service is to be performed is currently open. The two major "extraneous" objects are the Channel Control Block (CCB) and the File Control Block (FCB).

There are two CCB types: system CCBs and user CCBs. The main purpose of a CCB is to hold essential data that File Management and CCB Request Management use when servicing disk I/O requests. A unique system CCB is allocated from main system memory each and every time that AOS/VS implicitly opens a file, that is, without direction from the user. For example, when a user issues a ?CREATE system call on :DIR1:DIR2:FILE, the file system must open DIR1 in order to find DIR2, and open DIR2 in order to create the FNB/FIB pair for FILE. System CCBs are allocated for both DIR1 and DIR2 when they are opened and deallocated when they are closed; AOS/VS uses the CCBs to initiate I/O on the files. A user CCB is created for FILE in the user process' ring 1 and is used during ?READ and ?WRITE processing to initiate user I/O. Certain directory file management services must know about system CCBs.

An FCB is allocated from main system memory when a file is opened for the first time. The FCB is released when the file is closed for the last time. There is only one FCB for any given open file. The FCB contains a common area with the FIB called the Funny FIB (see Section 1.4.5). Depending upon the requested service, the relevant parameters will be changed in the FCB when the file is opened and flushed to the FIB when the file is closed. If the file is not open when the request is made, the FIB will be read into memory from disk, and later flushed if it was modified. The system CCB points to the FCB.

The FCB points to subordinate objects as well. One such pertinent object is the Control Point Block (CPB). The CPB contains a common area with the FIB which holds essential control point directory space information. If the file is open, the data will be retrieved/modified in the CPB and flushed to the FIB when the file is closed. If the file is not open when the service request is made, the same data in the FIB will be accessed. Since the CPB deals exclusively with directory management, it will be fully illustrated here:

Offset		Control Point Block (CPB)
CPCPB.W	0	Parent directory CPB address. Contains -1 if LDU root directory.
CPCSH.W	2	Current size of CPD.
CPMSH.W	4	Maximum size of CPD.
CPBLT	6	Length of CPB.

Since the FCB exists only when a file is open, the same is true for the CPB. Furthermore, if a file three directory levels deep is opened, AOS/VS must perform system opens on the superior directory files to do directory I/O. AOS/VS keeps the superior directory files open until the lowest level file is closed. This implies that FCBs and CPBs of the superior directories must be resident. The parent directory CPB address is needed because if the space in the lowest level directory changes, the space in the superior hierarchy changes as well. This space modification must be propagated up the hierarchy, and the CPBs of the superior directories are accessed via CPCPB.W. Current and maximum space requirements of open files are maintained in CPCSH.W and CPMSH.W.

2.3 DDB Allocation/Deallocation Operations

DDBs are physically allocated when AOS/VS performs an I/O request on a specified directory block. When a DDE is to be created, the DDB must be searched for enough free space to accommodate the DDE. For instance, when AOS/VS allocates a FIB, it must determine in which DDB to place it and where in the DDB to fit it. The execution of the ?CREATE system call adds a new file to a directory, which implies the potential allocation of new DDBs and the creation of a new FNB and FIB.

It is not the responsibility of ?CREATE code to perform these allocation operations, or the responsibility of ?DELETE code to perform DDB/DDE deallocation operations within its own code path. The AOS/VS file system provides modules that handle these procedures. These modules are named JELLO.P, JELUDA.P and REDEL. JELLO.P performs the operation of allocating DDBs (if necessary) and creating the input type of DDE (except FUD). JELUDA.P performs the operation of allocating a new DDB for a FUD. Both of these operations allocate the free DDE, but let the calling module appropriately fill it in. REDEL releases (frees up) a DDE.

2.3.1 JELLO.P: DDB Allocation and DDE Creation

JELLO.P performs the operation of allocating space in existing DDBs for a desired directory data element (except for DDE type FUD). JELLO.P searches the directory data blocks for enough space and reserves the space for the caller. If there is not sufficient space in any DDB for the new DDE, another DDB will be allocated and linked into the directory. JELLO.P passes back the address of the new DDE for the caller to fill it in appropriately.

The external modules that use the provisions of JELLO.P are:

Calling Module	Reason
CREATE.P	Allocate FNB at file creation time.
CREATE.P	Allocate FIB at file creation time.
CLINK.P	Allocate short FIB for link file.
CLINK.P	Allocate FLB for link file.
SACL.P/DOACL.P	Allocate FAC for file.
RENAME.P	Allocate FNB for renamed filename.

The inputs and outputs of JELLO.P are standard for all the calling modules:

Variable	Input	Output
AC0	Starting relative blk num to begin search.	DDB buffer header address.
AC1	Size in nuggets of desired DDE.	Unchanged.
AC2	Parent CCB address.	DDE address.

A typical calling sequence follows:

```
XNLDA 0,HASHV,3          ;SEARCH FROM FIB ROOT
XNLDA 1,DDESZ,3          ;DDE SIZE TO ALLOCATE
XWLDA 2,PCCB.W,3         ;PARENT CCB ADDR FROM STACK
LJSR  JELLO.P            ;ALLOCATE A FIB
  WBR JERR               ;HOW CAN JELLO BE BAD?

XWSTA 0,DDBBH.W,3        ;SAVE DDB BUFFER HEADER ADDR
XWSTA 2,FIBP.W,3         ;SAVE NEW FIB ADDR
```

If the DDE to be allocated is of type FNB, the starting DDB relative block number will resolve to the hash value of the file (the beginning of the FNB chain). If the DDE to be allocated is of type FIB, FAC or FLB, the starting DDB will be relative block number HFS (the FIB root). FUDs are not allocated by the JELLO.P operation.

The starting block number to search and the size in nuggets is sufficient to allocate the space for any valid DDE type. After all, FIBs can be either 2 or 4 nuggets long (020 or 040 words) and FLBs can be as long as 248. words. The caller need only request a specific DDE length. The parent CCB address is necessary for disk I/O.

JELLO.P returns the buffer header address of the DDB containing the new DDE. The caller must release and flush the system buffer (RELF) so that it will be immediately written to disk. The address of the DDE is also returned.

The following is the C-based pseudo-code algorithm that describes the JELLO.P operation:

```
#define NUGGET_SIZE 8
JELLO.P (start_blk, DDE_len, CCB_addr,          /* inputs  */
         *DDB_bh_addr_out,*DDE_addr_out)        /* outputs */
{
/*********************************************************
* Call BLKIN to read in starting block to search.        *
* Relative block number is supplied in the parent dir CCB.*
* BLKIN will allocate the block from disk (and store its  *
* logical disk address in the directory's index block) if *
* the DDB had not been previously allocated.  This case   *
* will occur if the DDE is the first FNB for its hash     *
* value.                                                  *
*********************************************************/

   DDB_rel_blk_num = CCB_addr->CBDBL;
   DDB_bh_addr = BLKIN (CCB_addr);
   DDB_addr = DDB_bh_addr->BQADR.W;

/*********************************************************
* Search for the desired free space.                     *
*********************************************************/

   DDE_addr = DDB_addr + NUGGET_SIZE;

   while (! end_of_DDB)
      {
      if (space_available >= DDE_len)
         {
         if (BLKIN_allocated_new_DDB)
            {
            set_bit (dir_bit_map, DDB_rel_blk_num);
            RELM (dir_bit_map_bh_addr);
            }

         *DDB_bh_addr_out = DDB_bh_addr;     /* Store     */
         *DDE_addr_out    = DDE_addr;        /* outputs.  */
         return;                             /* Return.   */
         }

       else
         DDE_addr = DDE_addr + NUGGET_SIZE;

      /*****************************************************
      * If the end of the DDB is reached without finding  *
      * enough space, read in the next DDB on the chain,  *
      * if it exists.  If not, the loop terminates anyway.*
      *****************************************************/

      if (end_of_DDB) && (DDB_addr->DENLB != 0)
         {
         CCB_addr->CBDBL = DDB_addr->DENLB;
         DDB_bh_addr = BLKIN (CCB_addr);
         DDB_addr = DDB_bh_addr->BQADR.W;
         DDE_addr = DDB_addr + NUGGET_SIZE;
         }
      }
```

```
    /*******************************************************
    * Search the bit map for a free DDB slot.  BLKIN will  *
    * allocate and read it in.                             *
    *******************************************************/

    DDB_rel_blk_num = search (dir_bit_map);
    CCB_addr->CBDBL = DDB_rel_blk_num;
    new_DDB_bh_addr = BLKIN (CCB_addr);
    new_DDB_addr = new_DDB_bh_addr->BQADR.W;

    DDE_addr = new_DDB_addr + NUGGET_SIZE;

    /*******************************************************
    * Link new DDB into directory structure.               *
    * FNB DDBs get linked to the end of its FNB chain.     *
    * General purpose DDBS get linked right after FIB root!*
    *******************************************************/

    if (FNB_DDB)
      {
      DDB_addr->DENLB = new_DDB_addr;
      new_DDB_addr->DELLB = DDB_addr;
      }
    else
      {
      /*****************************************************
       * Link in new DDB after FIB root.                   *
       * Read FIB root and its forward link DDB.           *
       * (The BLKINs are skipped here to avoid complexity) *
       * Then link the new DDB in between.                 *
       *****************************************************/

      FIB_root_addr->DENLB = new_DDB_addr;
      new_DDB_addr->DELLB = FIB_root_addr;

      FIB_root_flink_addr->DELLB = new_DDB_addr;
      new_DDB_addr->DENLB =FIB_root_flink_addr;
      }


   /*******************************************************
    * Set corresponding bit in bit map now.  It wasn't set *
    * before because there could have been errors reading  *
    * in the new DDB.  Finally, return with output values. *
    *******************************************************/

    set_bit (dir_bit_map, DDB_rel_blk_num);
    RELM (dir_bit_map_bh_addr);

    *DDB_bh_addr_out = new_DDB_bh_addr;      /* Store     */
    *DDE_addr_out    = DDE_addr;             /* outputs.  */
    return;                                  /* Return.   */
}
```

2.3.2 JELUDA.P: DDB Allocation and FUD Creation

JELUDA.P performs the operation of allocating new DDBs for DDE type FUD. JELUDA.P searches the directory bit map for a free DDB to hold the FUD. Since UDAs are 128. words long, the FUD has a 4 word header and the DDB has an 8. word header; a new DDB is allocated for each JELUDA.P request. However, the DDB is free for other types of DDEs after the initial FUD. JELUDA.P passes back the address of the FUD for the caller to fill it in.

The calling sequence to request the operation performed by JELUDA.P is "LJSR JELUDA.P". The only external module that makes this call is CRUDA.P. The input and output values in the accumulators are the same as those of a JELLO.P request (see Section 2.3.1).

Since the FUD is really just another DDE, the JELUDA.P algorithm is a subset of JELLO.P. The primary difference between FUD and other DDE type creation is that a new DDB is ALWAYS allocated for FUDs.

```
JELUDA.P (start_blk, DDE_len, CCB_addr,        /* inputs  */
          *DDB_bh_addr_new,*DDE_addr_new)      /* outputs */
{

/*************************************************************
* Search the bit map for an free general purpose DDB slot.*
* BLKIN will allocate and read in the DDB.                  *
*************************************************************/

   DDB_rel_blk_num = search (dir_bit_map);
   CCB_addr->CBDBL = DDB_rel_blk_num;

   new_DDB_bh_addr = BLKIN (CCB_addr);
   new_DDB_addr    = new_DDB_bh_addr->BQADR.W;
   DDE_addr        = new_DDB_addr + NUGGET_SIZE;

/*************************************************************
* Link new DDB into directory structure by making it the    *
* second DDB in the FIB chain (first after FIB root).       *
* Set corresponding bit in bit map now.  It wasn't set      *
* before because there could have been errors reading in    *
* the new DDB.  Finally, return with output values.         *
*************************************************************/

   link_in_new_DDB();                        /* See JELLO.P */

   set_bit (dir_bit_map, DDB_rel_blk_num);
   RELM (dir_bit_map_bh_addr);

   *DDB_bh_addr_out = new_DDB_bh_addr;       /* Store     */
   *DDE_addr_out    = DDE_addr;              /* outputs.  */
   return;                                   /* Return.   */
}
```

2.3.3 REDEL: DDB Deallocation and DDE Release

REDEL performs the operation of releasing the space occupied by a DDE. If the DDB holding the DDE becomes empty, it is unlinked from its current chain and the directory bit map is updated to show that the block is free. REDEL does not deallocate the block from the LDU bit map; the block remains unusable to the outside world. Its logical disk address is not deleted from the directory index block that points to it. So when another directory I/O operation requests the allocation of that same relative block number, the DDB will "still" be available. In other words, the LDU still believes that the block is allocated, so the only directory operations will be the reading in of the block from disk and the resetting of the bit in the directory bit map, marking the block as allocated. This saves time by avoiding extra disk block allocation procedure overhead. Incidentally, if the DDB is never reallocated by the directory and continues to be marked as "in use" by the LDU bit map, a run of FIXUP will clear the relative block number from the appropriate offset in the directory index block and clear the bit corresponding to the logical disk block in the LDU bit map.

The calling sequence to request the operation performed by REDEL is "LJSR REDEL". The external modules that make this call are the following:

Calling Module	Reason
CREATE.P	Release FNB DDE on error condition.
DELETE.P	Release FNB/FIB/FUD/FAC/FNB.
CLINK.P	Release FNB/FIB/FLB on error cond.
SACL.P/DOACL.P	Release old FAC before allocating new.
RENAME.P	Release FNB of renamed file.

The inputs and outputs of REDEL are standard for all the calling modules:

Variable	Input	Output
AC0	Parent CCB address.	Unchanged.
AC1	DDB buffer header address.	Unchanged.
AC2	DDE address.	Unchanged.

The DDB containing the DDE has already been read in by the caller. Its buffer header address must be passed to REDEL so the modified DDB can be flushed to disk (RELF) before the caller regains control.

The first word at the DDE address is the DDE's DETAS offset, which contains the type and length of the DDE. REDEL indicates that the DDE is free by zeroing out word 0 of each nugget that composes the DDE. JELLO.P checks only word 0 of a nugget to determine whether or not it is free; words 1 through 7 can be garbage.

Finally, the parent directory's CCB address is needed for disk I/O operations. There are no pertinent values to return.

The following is the C-based pseudo-code algorithm that describes the REDEL operation:

```
#define NUGGET_SIZE 8

REDEL (CCB_addr, DDB_bh_addr, DDE_addr,,,)  /* no output */
{
/*********************************************************
* Clear out all nuggets belonging to DDE.                *
* The size of the DDE is found in its DETAS offset.      *
*********************************************************/

   nuggets = (DDE_addr->DETAS & 0377) / NUGGET_SIZE;
   while (nuggets--)
      {
      DDE_addr->DETAS = 0;
      DDE_addr += NUGGET_SIZE;
      }

/*********************************************************
* Check to see if the DDB is now empty.                  *
* Do this by checking word 0 of each nugget for a 0 value.*
* If the DDB is not empty, all the work is done.         *
*********************************************************/

   DDB_addr = DDB_bh_addr->BQADR.W;        /* Top of DDB */
   nugget_ptr = DDB_addr + NUGGET_SIZE;    /* First DDE  */

   while (! end_of_DDB)
      {
      if (nugget_ptr->DETAS != 0)
         {
         RELF (DDB_bh_addr);
         return;
         }
      nugget_ptr += NUGGET_SIZE;
      }
```

```
/*********************************************************
* DDB is empty.                                          *
* If the DDB is not an anchor DDB, i.e. a root FNB/FIB   *
* DDB, unlink it from the DDB chain.                     *
* Do this by reading in the previous DDB and the next DDB *
* (if it exists) and updating their link words.          *
*********************************************************/

   DDB_rel_blk_num = DDB_bh_addr->BQDBN;

   if (DDB_rel_blk_num > hashframesize)  /* Anchor DDB?  */
      {                                  /* No           */
      CCB_addr->CBDBL = DDB_addr->DELLB; /* Read prev DDB*/
      prev_DDB_bh_addr = BLKIN (CCB_addr);
      prev_DDB_addr = prev_DDB_bh_addr->BQADR.W;

      if (DDB_addr->DENLB == 0)          /* Last DDB?       */
         {                               /* Yes, zero prev  */
         prev_DDB_addr->DENLB = 0;       /* DDB forward ptr */
         }                               /* to unlink DDB.  */

      else                               /* Not last DDB.   */
         {                               /* Read next DDB.  */
         CCB_addr->CBDBL = DDB_addr->DENLB;
         next_DDB_bh_addr = BLKIN (CCB_addr);
         next_DDB_addr = next_DDB_bh_addr->BQADR.W;

         /* Unlink the DDB: reset prev/next DDB pointers */
         prev_DDB_addr->DENLB = DDB_addr->DENLB;
         next_DDB_addr->DELLB = DDB_addr->DELLB;
         }
      }

/*********************************************************
* Whether unlinked or not, DDB is still empty.           *
* The bit in the directory bit map corresponding to the  *
* DDB's relative block number must be reset now.         *
* The bit map should be released modified since it was   *
* changed.                                               *
*********************************************************/

   clear_bit (dir_bit_map, DDB_rel_blk_num);
   RELM (dir_bit_map_bh_addr);

/*********************************************************
* Buffer Managament service RELF flushes the modified DDB *
* to disk.                                               *
*********************************************************/

   RELF (DDB_bh_addr);
   return;
}
```

2.3.4 RAID: Read a Directory Data Element

RAID is a Directory Management operation called by various File Management code paths that reads in a specific directory data element from a directory. For example, when RESLV.P wishes to read in a file's FIB, whose IDP it retrieved from the FNB, it calls RAID to accomplish the task. When RDUDA.P must read in the requested UDA, it calls RAID to read in the FIB, retrieves the FUD pointer from the FIB, and finally calls RAID to read in the FUD. The operation is simplistic. It just calls BLKIN, a File Management service, to read in the DDB which contains the DDE and calculates the requested DDE address.

The calling sequence to RAID is "LJSR RAID". The inputs and outputs of RAID are standard for all modules that use this operation:

Variable	Input	Output
AC0	IDP to desired DDE.	Unchanged.
AC1	DDE type requested.	DDB buffer header address.
AC2	Parent CCB address.	DDE address.

The relative directory block number is coded into the IDP, so RAID simply extracts it and calls BLKIN to read it in. The buffer header address must be returned to the caller, who is responsible for releasing the buffer. The DDE address is returned since that is what the caller requested.

The following is the C-based pseudo-code algorithm that describes the RAID operation:

```
RAID (DDE_IDP, DDE_type, PCCB_addr,                    /* Inputs */
      *DDB_BH_addr, *DDE_addr)                        /* Outputs */
{
  /*************************************************************
   * Shift IDP to get relative block number in directory.      *
   * Store it in the CCB and read in the block.                *
   *************************************************************/

   DDB_blk_num = DDE_IDP >> 5;
   PCCB_addr->CBDBH.W = DDB_blk_num;

   DDB_BH_addr = BLKIN (PCCB_addr);
   DDB_addr = DDB_BH_addr->BQADR.W;

  /*************************************************************
   * Ensure that the DDB just read in is really a DDB by       *
   * checking that the forward/back links are within range.    *
   *************************************************************/

   if (DDB_addr->DENLB < 0) || (DDB_addr->DENBL > 2047.) ||
      (DDB_addr->DELLB < 0) || (DDB_addr->DELLL > 2047.)
      {
      call PNIC (6031);                    /* Invalid DDB     */
      }

  /*************************************************************
   * Get the DDE address from the offset field of the IDP.     *
   * If the type read does not match the type request, panic.  *
   *************************************************************/

   DDE_addr = DDB_addr + (DDE_IDP & OFFSET_MASK) * NUGGET_SIZE;
   if (get_DDE_type (DDE_addr->DETAS) != DDE_type)
      {
      call PNIC (6032);                    /* Invalid DDE     */
      }

  /*************************************************************
   * If the DDE size is not a multiple of 8 (nugget size),     *
   * probably not a valid DDB or DDE.  So, blow up the world.  *
   *************************************************************/

   if (get_DDE_size (DDE_addr->DETAS) != multiple_of_NUGGETSIZE)
      {
      call PNIC (6014);                    /* Invalid DDE     */
      }

  /*************************************************************
   * DDE OK.  Parameters already stored for caller.  Bye.      *
   *************************************************************/

   return;

} /* end of RAID */
```

2.4 Directory Management Services

2.4.1 Pathname Resolution Services

File Resolution Services performs the function of resolving an input pathname, which may contain multiple filenames separated by colons and which may be pre-pended with a valid prefix. Pathname "resolution" covers much more than merely checking for the existence of each file supplied in the pathname. In order for the file system to determine if a filename is present in a directory, the directory must be opened (implying that a system CCB and FCB must be created) and the FNB chain in which the file would be found must be searched. If the file is found, its FIB must be read. If the file is not the last file in the pathname, its starting address must be obtained, it must be opened, searched, etc., until the last filename of the pathname is reached. All files previous to the last filename must be directories. Depending upon the variant of resolution service that was requested, some specific action will be taken. The most basic and common variant of resolution services, RESLV.P, is called by system call processing modules whose input is a file pathname.

There are five File Resolution Services variants. Each variant and a functional description of the service it provides is listed below:

Variant	Function
RESLV.P	Search for the existence of the files named in the input pathname.
WRSLV.P	Same as RESLV.P, except the last file in the pathname must not exist.
DRSLV.P	Same as RESLV.P, except the last file in the pathname must be a directory.
GRSLV.P	Same as RESLV.P, except special handling for ?GNAME call. Uses current searchlist.
RRSLV.P	Same as RESLV.P, except special handling for ?GFNAME call. Uses input PID's searchlist.

The inputs to all variants of resolution services are standard. The caller supplies the required information in the accumulators as follows:

Variable	Input	Output
AC0	Byte address of file pathname.	Depends on variant.
AC1	Input switches.	Depends on variant.
AC2	Address flag (bit 0), Address of initial search directory CCB.	Depends on variant.
"OAC4"	Not used.	Depends on variant.
"OAC5"	Not used.	Depends on variant.

The input contents of AC0 is always a byte pointer to the pathname of the file to be resolved. The pathname will be in user space if input to a system call. The pathname will be in system space if the input pathname was generated by the system (breakfile creation, system initialization calls). Bit 0 of AC2 must be set if the pathname is already in system space.

The input switches are passed in AC1. These bits represent options that will be considered in resolving the pathname. The input switches corresponding to the following bits are:

Bit 0) Set: do not resolve links occurring in pathname.
Reset: resolve links occurring in pathname.

Bit 1) Set: return 0 in OAC5 if pathname has prefix "@".
Reset: do not alter the contents of OAC5.

Bit 27) Set: if a non-directory argument is found in the pathname during GRSLV.P, it must be a console.
Reset: non-directory pathname arguments need not be a console.

Bit 30) Set: if directory access denied error, check for complete C2 logging and record pathname of dir with error and the address of buffer passed back in AC2 on error return.
Reset: do not check for complete logging.

Bit 31) Set: do not apply search rules.
Reset: apply search rules.

Sometimes the caller may not wish to resolve links. The CLI selects this option when it makes the ?FSTAT system call. Notice that when doing a "FILESTATUS" from CLI with a link filename embedded in the pathname, a "File does not exist" error will be produced even if the file exists!

Some code paths must know whether or not the input pathname contained a prefix. For example, if the user supplies the pathname for a breakfile either via ?BRKFL, ?MDUMP or ?TERM, and the destination directory is @ (:PER), the breakfile creation will be aborted if the file already exists. The caller of WRSLV.P in this case must check for this condition.

There is a special feature of ?GNAME that permits the system call to verify the existence of a console device on the input pathname. ?GNAME will call GRSLV.P with bit 27 of AC1 set, and the file type of the first non-directory filename in the pathname will be verified for a console type file. This feature is provided for AOS/VS windowing support.

If C2 logging is enabled on the system, access violations will be logged. If bit 30 of AC1 is set, complete logging will be done if "Directory access denied" errors are incurred during pathname resolution.

Search rules are applied if the caller's searchlist is to be used in finding the file. For example, search rules are applied when ?CRUDA utilizes RESLV.P to resolve the input pathname and ultimately create the file's UDA. If RESLV.P does not find the file in the caller's current working directory, the searchlist CCBs in the caller's process table extender will be accessed, and the desired file searched for in each of those directories. Search rules are not applied, for example, on file deletion.

Finally, the initial search directory CCB address may be supplied in AC2. If the caller wishes the initial search directory to default to the current working directory, AC2 is zeroed out.

The values returned to callers of Resolution Services routines depend upon the requested variant. There are potentially more than three values to return. Callers must provide for two extra stack variables, called OAC4.W and OAC5.W, which are pushed on the stack before the LJSR to RESLV.P (or variant). This stack space provides two double words reserved for output values to be returned to the caller (if necessary).

In order to fully comprehend the reasoning of returning specific outputs, you should be familiar with the modules that request the services, which are called primarily from modules contained within the data management component of AOS/VS (i.e., the file system). There are, however, external components that make use of the services. The following tables list the modules that call each separate variant and the returned output values that the callers will use.

External requesting modules of RESLV.P

Caller	Effect
CRUDA.P DELETE.P FSTAT.P GACL.P GLINK.P GTACP.P ILKUP.P LINK.P OPEN.P RDUDA.P RENAME.P RNAME.P SACL.P SATR.P WRUDA.P	These system call services require that the caller supply a pathname on which a desired function is to be performed. They all call RESLV.P to resolve the input pathname.

Variable	Output (input pathname not JUST a prefix)
AC0	FIB pointer (IDP) of last file in pathname.
AC1	FNB pointer (IDP) of last file in pathname.
AC2	Parent directory CCB pointer.
"OAC4"	File number (if unit file) or -1.
"OAC5"	0 (if prefix is "@") or unchanged.

Variable	Output (input pathname JUST a prefix)
AC0	Byte pointer to end of pathname + 1.
AC1	0.
AC2	Directory's CCB pointer.
"OAC4"	-1.
"OAC5"	0 (if prefix is "@") or unchanged.

RESLV.P is called by code paths that implement system calls with inputs that include a file pathname. Each filename in the pathname is validated for existence and valid file type, including the last file. For example, if RESLV.P ever comes across a file of type ?FMDR that is valid only under RT32, a system panic (code 6023) will be forced. Each file in the pathname that is not the last file must be a valid directory type file. The last file may be any valid AOS/VS file type.

Since resolution of the last filename is the purpose of this service, the caller will require either the file's FNB or its FIB, or both. The IDPs are passed back so the caller can decide which DDE to read in. RENAME.P calls RESLV.P to request the FNB and FIB pointers of the existing filename. It must read in the file's FNB (later to delete it) and its FIB (to adjust the existing FNB pointer to the FNB of the new filename). On the other hand, SACL.P only needs the FIB to access the FAC; the FNB pointer is not used. The parent CCB address is used by all callers of Resolution Services for access to the parent directory's databases and to initiate directory I/O; the FNB of any file resides in its parent directory.

The special case of the pathname consisting solely of a prefix (":", "^", "^^^", "=", "@") does not necessitate the output of the same values as the case of the pathname consisting of filename characters. A prefix must resolve to a directory type file; therefore, the directory's CCB is an adequate parameter to return. The CCB already contains the FIB IDP (CBFIB), and the directory's FNB is accessible through the FIB. OAC5 will be set to 0 if the prefix is "@", since BRKFL.P needs to know this information (see WRSLV.P).

A typical calling sequence to RESLV.P is illustrated by the following call from ILKUP.P:

```
        WSUB   1,1              ; APPLY SEARCH RULES
        WSUB   2,2              ; PATHNAME IN USER SPACE
        WPSH   1,2              ; MAKE ROOM FOR OAC4,5
        XWLDA  0,TAC0.W,2       ; GET THE USER'S AC0
        LJSR    RESLV.P         ; RESOLVE THE PATHNAME
          WRTN                  ; ERROR IN CERWD OF CB

        WNEZ   1                ; PATHNAME JUST PREFIX?
        WBR    DIERR            ; YES - IT CAN'T BE IPC FILE
        XWSTA  2,PCCB.W,3       ; REMEMBER THE PARENT DIR
        NLDAI  DEFIB,1          ; FIB = ELEMENT TYPE REQUESTED
        XJSR   RAID             ; READ IN THE FIB
          WBR  BERR             ; RATS!
```

External requesting modules of WRSLV.P

Caller	Effect
BRKFL.P	Resolve pathname of breakfile to create.
CREATE.P	Resolve pathname of file to create.
RENAME.P	Resolve pathname of new filename.

Variable	Output
AC0	Hash value of the last filename.
AC1	Namespace address containing last filename.
AC2	Parent directory CCB pointer.
"OAC4"	Byte length of last filename, or host id.
"OAC5"	0 (if prefix is "@") or unchanged.

The WRSLV.P variant of Resolution Services is called when the last filename of the supplied pathname does not exist. This implies that the caller plans to create the file, which is indeed the reason that WRSLV.P is chosen. CREATE.P, RENAME.P and BRKFL.P utilize this service to verify the validity of the directory in which the new file is to be created and to acquire essential information leading to the file's creation.

Although BRKFL.P calls BCREATE.P, which calls WRSLV.P, BRKFL.P must "pre-resolve" the breakfile name; if the breakfile name already exists, it must be deleted and re-created. Furthermore, since BRKFL.P does not want to delete any file in :PER, the preliminary call to WRSLV.P prevents this condition from arising. BRKFL.P will check the returned value in OAC5 for 0, which would indicate that the parent directory of the existing file is :PER. If this is true, no breakfile will be created.

WRSLV.P returns parameters that the caller will need to create a file. The caller uses the hash value as input to JELLO.P when creating the FNB. The namespace, a buffer in system memory containing the last filename in the pathname, is used to fill in the FNB. The filename length is used as an FNB parameter as well. As usual, the parent CCB address is always passed back.

Incidentally, WRSLV.P returns the host id of a file in the pathname that is of type ?FREM, along with error code "Remote resource reference made."

External requesting modules of DRSLV.P

Caller	Effect
CPMAX.P	Resolves CPD directory pathname.
DRLSE.P	Resolves input LDU pathname.
GFTLDU.P	Resolves directory name to graft initted LDU onto.
MIRROR.P	Resolves input LDU pathname.
PROC2.P	Resolves initial process working directory.
SINIT1	Resolves :PER and :NET directories to save their system CCB addresses.
SLIST.P	Resolves input searchlist directory.

Variable	Output
AC0	Byte pointer to end of pathname + 1.
AC1	Unchanged.
AC2	Directory's CCB pointer.
"OAC4"	Unchanged.
"OAC5"	Unchanged.

DRSLV.P is called when the last filename in the pathname is a directory. The purpose of this call is actually to create and/or retrieve the system CCB address of the directory, i.e., open the directory. For example, system initialization must call DRSLV.P to create system CCBs for both :PER and :NET, storing them on the return in PERCID.W and NETCID.W, respectively. DRLSE.P requests the system CCB of the input directory (LDU) to retrieve the system CCB that was created on the LDU's initialization. The return value in AC0 is not used by any of DRSLV.P's callers. The two double words, OAC4 and OAC5, need not be pushed before calls to DRSLV.P, since they are neither accessed nor altered.

External requesting modules of GRSLV.P

Caller	Effect
GNAME.P	Resolves input pathname.
LINK.P	Resolves pathname found in file's FLB.

External requesting modules of RRSLV.P

Caller	Effect
GNAME.P LINK.P	Resolves input pathname. Resolves pathname found in file's FLB.

Variable	Output
AC0	FIB IDP to last disk file in pathname.
AC1	FNB IDP to last disk file in pathname.
AC2	Parent directory CCB pointer.
"OAC4"	Address of namespace (if unit file), or -1. If last file's type is ?FREM, return host id.
"OAC5"	Num unused bytes in namespace, or unchanged

GRSLV.P is the customized resolve for the ?GNAME system call, and RRSLV.P is the customized resolve for the similar ?GFNAME call. While ?GNAME resolves the whole pathname (resolving links) given an input pathname and the caller's searchlist, ?GFNAME does the same using the searchlist of the PID of one of its customers (the caller must be a server).

GRSLV.P searches for the existence of the files named in the pathname. If any file in the pathname is a physical or logical unit, a network type file (except ?FREM) or a peripheral console device, GRSLV.P passes back the remaining files of the pathname. These files are unverified, accessible through the namespace pointer in OAC4. If bit 27 is set in AC1, the first Non-directory filename's file type is assumed to be type ?FCON, and an error is returned if it is not ("Non-directory argument in pathname"). This feature is used by AOS/VS windowing calls to retrieve and verify the console filename from a pathname. If the input pathname contains a network file of type ?FREM and it is not the last file, the host id will be returned along with the error "Illegal host specification." The AGENT will act upon this condition and deflect the call to RMA if the host specification is indeed legal. If there are no errors, the last verified disk file's FNB and FIB pointers and its parent CCB address are returned. The namespace length is a constant 128. words; if the remaining number of bytes in the pathname do not occupy 128. words, the number of unused bytes in the namespace is returned in OAC5.

RRSLV.P works the same way as GRSLV.P, except that it treats all network files as logical unit or console files. There is no exception for ?FREM type files.

Observe the following CLI command lines and outputs:

```
)PATHNAME :LINK_TO_DIRX:FILE
:DIRX:FILE

)PATHNAME :UTIL:XHELP.CLI:OH_NO
Warning: Non-directory argument in pathname

)PATHNAME :PER:MTB0:0:1:2:THIS:IS:UNVERIFIED
:PER:MTB0:0:1:2:THIS:IS:UNVERIFIED

)PATHNAME :NET:RMA:NPN:FILES:UNVERIFIED
:NET:RMA:NPN:FILES:UNVERIFIED
```

Since RESLV.P is called most frequently, its C-based pseudo-code algorithm will be illustrated here.

```
RESLV.P (*pathname, input_switches, OAC2.W,
         *FIB_IDP, *FNB_IDP, *PCCB_addr, *file_num, *pref_flag)
{
/*************************************************************
 * Initialization.                                           *
 * Init file num to -1.  Extract info from aflag.            *
 *************************************************************/

   *file_num = -1;

   *PCCB_addr = OAC2.W & 017777777777;
   pathname_in_system = OAC2.W & 020000000000;

/*************************************************************
 * If pathname is in user space, map it to system space.     *
 * If the pathname spans a page boundary, use the dynamic    *
 * logical slots to map the two logical pages contiguously.  *
 * If the pathname falls on one page only, fault and pin it. *
 *************************************************************/

   if (!pathname_in_system)
      {
      RMAPU (num_wds_to_map, (int *) pathname, CC.W->CPTAD.W);
      if (error)
         return (Invalid_word_pointer);
      }
```

```
/*************************************************************
 * Check initial search directory CCB.                       *
 * If none specified, determine what it should be and        *
 * temporarily set the parent CCB address to return.         *
 * The initial working dir is located in the process table   *
 * extender.                                                 *
 *************************************************************/

TOP:
   WDIRCCB_addr = CC.W->CPTAD.W->PEXTN.W->PWCCB.W;

   if (*PCCB_addr == 0)
      {
      prefix_found = TRUE;                    /* Assume prefix   */
      switch (prefix = *pathname++) {
         case '@':                            /* Start with :PER */
            *pref_flag = 0;
            *PCCB_addr = PERCID.W;
            break;

         case '^':                            /* Up dir hierarchy*/
            *PCCB_addr = WDIRCCB_addr->CBPCB.W;
            while (*pathname++ == '^')
               *PCCB_addr = *PCCB_addr->CBPCB.W;
            break;

         case ':':                            /* Start with root */
            *PCCB_addr = RTCCB;
            break;

         case '=':                          /* = means working dir*/
            *PCCB_addr = WDIRCCB_addr;
            break;

         case default:                      /* Default working dir*/
            *PCCB_addr = WDIRCCB_addr;
            prefix_found = FALSE;           /* Whoops - no prefix */
         }
      }

/*************************************************************
 * Return now if pathname ONLY a prefix.                     *
 * The return values for this case differ from the typical   *
 * case of more than just a prefix.  See inputs/outputs above.*
 * Release the user pages from the DLS as well.              *
 *************************************************************/


   if (prefix_found && end_of_pathname())
      {
      *OAC0.W = pathname;
      *OAC1.W = 0;
      if (!pathname_in_system)
         UNMAP (num_wds_mapped,0,(int *) pathname);
      return;
      }
```

```
/*************************************************************
 * Allocate namespace where isolated files in the pathname    *
 * will be saved.  Then resolve pathname.                     *
 *************************************************************/

   namespace_ptr = GSMRS (FNSSZ);

   while (!end_of_pathname())
      {
      /*********************************************************
       * This while loop checks for the existence and the      *
       * validity of each file in the pathname.  Directory     *
       * files in the pathname will essentially be opened by   *
       * the system, manifested by the creation of an FCB and  *
       * system CCB.  Non-directory files will be tested for   *
       * legality with various criteria.  Upon input to this   *
       * loop, *PCCB_addr actually points to the current dir,  *
       * but will later become the parent dir pointer of the   *
       * last file in the pathname, a returned parameter.      *
       *********************************************************/

      /*********************************************************
       * Check: execute access to directory allowed.           *
       *********************************************************/

      if (error = ESTAC.P (PCCB_addr, &ACL_privs))
         return (error);

      if (ACL_privs & execute_access) == 0)
         return (directory_access_denied);

      /*********************************************************
       * Copy (next) filename in pathname to namespace and     *
       * check for invalid characters.                         *
       *********************************************************/

      namespace = namespace_ptr;
      while (legal_filename_char(*pathname))
         *namespace++ = *pathname++;

      if (*(namespace-1) == ':' || legal_pathname_terminator())
         *namespace++ = NULL
      else
         return (illegal_filename_character);

      /*********************************************************
       * Find the filename in the directory.                   *
       * The LOOKUP.P operation searches the FNB chain in      *
       * which this file would be found.  It returns the FNB   *
       * and FIB pointers if the file is found.                *
       * If found, read in the file's FIB to proceed.          *
       *********************************************************/
```

```
if (error = LOOKUP.P (hash_value(*namespace_ptr),
                      (int *) namespace, *PCCB_addr,
                      &FIB_IDP, &FNB_IDP))
   return (file_not_found);

if (error = RAID (FIB_IDP, DEFIB, *PCCB_addr,&FIB_bh_addr)
   return (error);

FIB_addr = FIB_bh_addr->BQADR.W;

/*******************************************************
 * Check file types.                                   *
 * Link files must be resolved by LINK.P.  We chain to *
 * LINK.P, who will do this work.  This is done if the *
 * resolve links switch was set on input AC1.          *
 * LINK.P returns with the new PCCB_addr!              *
 *******************************************************/

if (FIB_addr->FITYP == link_type)
   if (input_switches & NRESLNK) && (!end_of_pathname)
      chain (LINK.P);
   else
      break; /* done */

/*******************************************************
 * If end of pathname, RESLV.P is done!                *
 * All return parameters have already been obtained.   *
 *******************************************************/

if (!end_of_pathname())
   {
   /*****************************************************
    * RESLV.P resolves ?FHST files (via NRSLV.P).       *
    * Only DRSLV/RRSVL resolves other network types.    *
    * By the way, ?GNAME does a DRSVL!                  *
    *****************************************************/

   if (FIB_addr->FITYP == ?FHST)
      chain (NRSLV.P);

   /*****************************************************
    * Examine the pathname to get the unit file number. *
    * This will mark the end of the pathname.           *
    *****************************************************/

   else if (FIB_addr->FITYP == unit_type)
      *file_num = get_file_number (pathname);
```

```
/*****************************************************
 * Normally, middle of pathname files are dirs.      *
 * AOS/VS will OPEN the directory.                   *
 * Basically, create an FCB if not already open, slam*
 * its address into the FIB, and release the FIB.    *
 * The parent CCB use count will be incremented with *
 * the creation of a new FCB (a "son").  The file    *
 * open count (FCB_addr->FBOPN) is not incremented,  *
 * but the system CCB addr will indicate the system  *
 * "implicitly" has the file open.                   *
 *****************************************************/

else if (FIB_addr->FITYP == directory_type)
   {
   FFCB (*PCCB_addr, FIB_IDP, &FCB_addr); /* Get FCB */

   if (FIB_addr->FIFCB.W == 0)
      {
      FIB_addr->FIFCB.W = FCB_addr;
      RELM (FIB_bh_addr);        /* Rel FIB modified */
      }
   else
      RELB (FIB_bh_addr);      /* Rel FIB unmodified */

/*****************************************************
 * Create a system CCB if it does not already exist. *
 * When done, update the current parent CCB pointer  *
 * and return to the top of the while loop to cont.  *
 * and open the next filename in the pathname.       *
 * The parent CCB use count must be incremented since*
 * a new son CCB was created and POINTS TO IT!       *
 * The use count is decr when the son CCB released.  *
 *****************************************************/

   if ((SCCB_addr=FCB_addr->FBSCB.W) == 0)
      {
      /* No system CCB: create it, lock it, init it */
      SCCB_addr = GSMRS (CCBLT);
      CLOCK (SCCB_addr);
      SCCB_addr->CBFCB.W = FCB_addr;
      SCCB_addr->CBPCB.W = PCCB_addr;
      FCB_addr->FBSCB.W  = SCCB_addr;
      SCCB_addr->CBUID   = FCB_addr->FBUID;
      PCCB_addr->CBUSC   = 1;
      }
   else
      /* Dir already open, CCB exists.  Just lock it */
      CLOCK (SCCB_addr);

/*****************************************************
 * Unlock current parent CCB.                        *
 * Update the PCCB_addr to point to the current dir. *
 *****************************************************/

   CULCK (PCCB_addr);
   PCCB_addr = SCCB_addr;
   }

else
```

```
            /************************************************
             * ILLEGAL FILE TYPE FOUND WITHIN PATHNAME!      *
             * If search rules, get next searchlist CCB and  *
             * start all over.  Else, error out.             *
             ************************************************/

            if (input_switches & search_rules)
               {
               *PCCB_addr = get_next_searchlist_CCB();

               if (*PCCB_addr == 0)                  /* No more */
                  return (Non-directory_argument_in_pathname);
               else
                  {                              /* Look in new dir! */
                  KCCB.P (old_PCCB_addr); /* Release old CCB  */
                  CLOCK (*PCCB_addr);     /* Lock up new CCB  */
                  goto TOP;               /* Go to stage 1    */
                  }
               }
            else
               /* No search rules ... no directory */
               return (Non-directory_argument_in_path);

         } /* if */
      }  /* while */

   /************************************************************
    * DONE!                                                    *
    * We have resolved the pathname.  The FNB/FIB pointers     *
    * have been set by LOOKUP.P, and RESLV.P has already       *
    * taken care of the rest.  Let the caller do as it wishes  *
    * with the data.                                           *
    ************************************************************/

   RSMEM (FNSSX, namespace_ptr);
   RELB (FIB_bh_addr);
   if (aflag == 0)
      UNMAP (num_wds_mapped, 0, (int *) pathname);

   return;
}
```

2.4.2 File Creation Services

File Creation Services performs the function of creating a file, specifically, of creating a unique FNB and a FIB in the file's parent directory. If an ACL is provided, a FAC will be created. If the file is of type link, an FLB will be created. Since there are numerous FIB parameters redefined for various file types, the FIB contents will vary.

There are five variations of File Creation Services. Each variant is accessed through a specific entry point. The different types of "creates" and their functionality are the following:

Variant	Function
CREATE.P	Create a file for a user. Implementation of the ?CREATE system call.
ICREATE.P	Create a file for the system.
VCREATE.P	Create an LDU entry in a directory.
BCREATE.P	Create a breakfile.
PCREATE.P	Create per process swap/page files.

External requesting modules of CREATE.P

Caller	Effect
User	?CREATE - Create a file.
SINIT1	Create :BOTH:SWAP and :BOTH:PAGE links if :BOTH LDU hosts SWAP and PAGE directories
SINIT1	Create actual SWAP and PAGE directories.

Note: ICREATE.P is not designed to create links; system initialization must call CREATE.P to accomplish this. CREATE.P is called instead of ICREATE.P to create the SWAP and PAGE directories because a maximum CPD size must be specified; ICREATE.P cannot do this for CPD types.

External requesting modules of ICREATE.P

Caller	Effect
SINIT1	Create :PER and :NET directories
SINIT1	Create @CONSOLE; generic files; unit files; :PROC:HIF, :PROC:PIF, :PROC:IPS.00003 file (IPC spool file for CLIBT); @LFD.

Note: :PER and :PROC are always deleted and re-created. ICREATE.P is always called to create :NET during system initialization; however, if an error code is returned, the code is assumed to be ERFAE, "File already exists."

External requesting modules of VCREATE.P

Caller	Effect
XINIT.P	Graft LDU entry into directory hierarchy during disk initialization.

External requesting modules of BCREATE.P

Caller	Effect
BRKFL.P	Create a breakfile.

External requesting modules of PCREATE.P

Caller	Effect
SWAPFILES	Create per process page file. Create per process swap file.

Most Directory and File Management system calls run the risk of pending on I/O completion. CREATE.P and its variants are not exceptions. The callers of file creation services must be running on a control block or a daemon so that state save data may be stored safely. System initialization creates a primary control block for its own use and allocates a dummy TCB in order to issue standard system calls that may pend, including CREATE.P. Errors incurred during processing will be returned to the caller in AC0.

The ?CREATE system call, accessed via the CREATE.P entry point, must transfer packet data from the caller's address space to system space. This is done with a WBLM instruction. However, it is entirely possible that an access violation could occur and cause a trap. CREATE.P handles this potential danger by establishing its own fault handler, CTRP, whose address is stored in the control block at offset CBFEH.W, before touching memory in the caller's address space. If a fault occurs, control will be transferred to CTRP, the error condition will be analyzed, and CREATE.P will decide what course of action to follow. If the trap code indicates that the control block is still valid, the error code will be returned to the caller. If the control block is invalid, but the error condition is a memory restart, the call will be restarted. However, if the control block is invalid and the trap code is not a restart, it means that the hardware has detected some unknown protection violation, and CREATE.P will panic the system with panic code 7304. This is general system call handling and will not be repeated in the following sections.

The input values to each create variant are dependent upon the variant being called. These parameter differences are screened and arranged so that the flow of control can continue at a common point for all variants. The most common service requested is that of CREATE.P, presented here in algorithm form.

The inputs and outputs of CREATE.P are listed below. TAC n refers to accumulator n in the calling TCB.

Variable	Input	Output
TAC0.W	Byte address of file pathname in user space	Unchanged.
TAC1.W	Not used.	Unchanged.
TAC2.W	Address of packet.	Unchanged.

If CREATE.P is being accessed by a user process, the standard system call interface is the calling sequence. The only other component that makes use of CREATE.P is system initialization. A typical implementation follows:

```
        XWLDA   2,TCBAD.W,3     ;RETRIEVE TCB ADDR
        XLEF    0,LNKPK         ;PACKET ADDRESS
        XWSTA   0,TAC2.W,2      ;SET UP PACKET ADDRESS FOR CALL
        LLEFB   0,PGNAM*2,0     ;BP TO :PAGE
        XWSTA   0,TAC0.W,2      ;SET UP PATHNAME FOR CALL
        LJSR    CREATE.P        ;CREATE LINK TO :BOTH:PAGE
          WBR   PRINTERR        ;PRINT THE ERROR CODE
```

CREATE.P is the only variant that takes a packet as an input parameter. The user supplies the required information for file creation in the create packet, which is moved into system space and subsequently extracted as needed. When the file is finally created, there is nothing pertinent to return to the caller, except a successful or failure status value. The following is the C-based pseudo-code algorithm that describes the CREATE.P service:

```
CREATE.P (*pathname,,*caller_pkt)        /* No outputs        */
{
/*************************************************************
 * Move caller's CREATE.P packet to system space.             *
 * AOS/VS does this with a WBLM instruction.  If the caller   *
 * supplied a time block, it is also moved to system space.   *
 *************************************************************/

   wblm (caller_pkt, &sys_pkt, pkt_len);

/*************************************************************
 * Check validity and resolve pathname.                       *
 * If the file type is illegal, that is, not within the       *
 * file range or not found in the LFTBL (legal file table),   *
 * return the error.  Otherwise, check for the presence of    *
 * each dir supplied in the pathname by calling WRSLV.P.      *
 * The parent CCB is needed to access the parent dir's FCB!   *
 *************************************************************/

   if (sys_pkt->cftyp < ?SMIN) || (sys_pkt->cftyp > ?SMAX) ||
      (check_bit (LFTBL_addr, sys_pkt->cftyp))
      return (illegal_file_type_error);

   call WRSLV.P (pathname, switches, a_flag,
                 &hash_val, &filename, &PCCB_addr);
```

```
/*************************************************************
 * Check directory access.                                   *
 * Caller must have write or append access to create the file.*
 * Then, more validity checking is done.                     *
 *************************************************************/

   call ESTAC.P (PCCB_addr, &ACL_privs);

   if (ACL_privs & (write_access | append_access) == 0)
      return (directory_access_denied);

   /* Must create network files in :NET */
   if (network_type_file && (PCCB_addr != NETCID.W)
      return (illegal_dir_name_specification);

   /* Cannot exceed max dir depth, currently 8 */
   PFCB_addr = PCCB_addr->CBFCB.W;
   if (PFCB_addr->FBLVL == SCLVL)
      return (max_dir_tree_depth_exceeded);

/*************************************************************
 * !!! CREATE AN FNB IN THE PARENT DIRECTORY !!!.            *
 * The only error can be insufficient room in directory.     *
 *************************************************************/

   call JELLO.P (hash_val, filename, PCCB_addr,
                 &FNB_bh_addr,&FNB_addr);

   if (error) return (insufficient_room_in_dir);

/*************************************************************
 * Fill in the DETAS word and move the filename into the FNB. *
 * The FIB must be created before its pointer is stored.     *
 *************************************************************/

   FNB_addr->DETAS = (DEFNB << 8) + strlen(filename) + 2;
   call round_up_to_next_nugget_multiple (FNB_addr->DETAS);

   wblm (&filename, &FNB_addr->FNNAM, strlen(filename);


/*************************************************************
 * Let Link Services create the FIB and FLB.  Chain away!    *
 *************************************************************/

   if (link_type_file)
      chain (CLINK.P);

/*************************************************************
 * !!! CREATE A FIB IN THE PARENT DIRECTORY !!!              *
 * The FIB contents depend upon file type.                   *
 * No error conditions will arise after FIB allocation.      *
 *************************************************************/

   call JELLO.P (hash_val, filename, PFCB_addr->FBHFS,
                        &FIB_bh_addr,&FIB_addr);

   if (error) return (insufficient_room_in_dir);
```

```
switch (sys_pkt->cftyp) {
   case IPC_FILE_TYPE:

     /*******************************************************
      * Fill in the global port number.                     *
      *******************************************************/

      FIB_addr->DETAS = FDETAS;          /* Full length FIB */
      FIB_addr->FIPHI = pid;
      FIB_addr->FIPHL = ring_and_local_port;
      break;

   case GENERIC_FILE_TYPE:

     /*******************************************************
      * The generic file indices are as follows:            *
      * 1=@INPUT, 2=@OUTPUT, 3=@LIST, 4=@DATA, 5=@NULL      *
      *******************************************************/

      FIB_addr->FICPS = generic_file_index;
      break;

   case UNIT_FILE_TYPE:

     /*******************************************************
      * Fill in the unit's device code and number.          *
      * Fill in mag tape redefs if mag tape unit.           *
      * Units have no starting address at creation.         *
      *******************************************************/

      FIB_addr->FIDCU = (device_code << 8) + unit_number;

      if (sys_pkt->cftyp = ?FMTU)
         init_tape_redef_params_in_FIB (FIB_addr);

      FIB_addr->FISTS = 0;
      FIB_addr->FIFAH.W = 0;
      FIB_addr->FIFCB.W = 0;
      break;

   case DIRECTORY_TYPE_FILE:

     /*******************************************************
      * Fill in directory-related FIB parameters.           *
      *******************************************************/

      FIB_addr->FIMSH.W = sys_pkt->cmsh;     /* Max space */
      FIB_addr->FICSH.W = 0;                /* Curr space */
      FIB_addr->FIHFS   = sys_pkt->chfs;           /* HFS */
      FIB_addr->FIIDX   = SCMIL;          /* Max index lev */
      FIB_addr->FIDFH.W = 1;              /* elementsize=1 */

      /* Time last accessed */
      FIB_addr->FITAH.W =
          ((sys_pkt->CTIM == -1) ? system_time : user_time);
```

```
        /* Time last modified */
        FIB_addr->FITMH.W =
            ((sys_pkt->CTIM == -1) ? system_time : user_time);

        FIB_addr->FIFUD   = 0;                  /* Init the */
        FIB_addr->FIEFH.W = 0;                  /* rest to  */
        FIB_addr->FIFW1   = 0;                  /* zippo.   */
        FIB_addr->FIFW2   = 0;
        FIB_addr->FIIDR   = 0;
        FIB_addr->FISTS   = 0;
        FIB_addr->FIFAH.W = 0;
        break;

    case default:

      /*******************************************************
       * For all other file types, fill in the FIB with     *
       * values supplied in the packet, or init field to 0. *
       *******************************************************/

        FIB_addr->FICPS   = sys_pkt->FICPS;
        FIB_addr->FIIDX   = sys_pkt->cmil;
        FIB_addr->FIDFH.W = sys_pkt->cdel;

        /* Time last accessed */
        FIB_addr->FITAH.W =
            ((sys_pkt->CTIM == -1) ? system_time : user_time);

        /* Time last modified */
        FIB_addr->FITMH.W =
            ((sys_pkt->CTIM == -1) ? system_time : user_time);

        FIB_addr->FIFUD   = 0;
        FIB_addr->FIEFH.W = 0;
        FIB_addr->FIFW1   = 0;
        FIB_addr->FIFW2   = 0;
        FIB_addr->FIIDR   = 0;
        FIB_addr->FISTS   = 0;
        FIB_addr->FIFAH.W = 0;
    }

/*************************************************************
 * Fill in other FIB offsets.                                *
 * If a time block was specified in the packet, use the time *
 * specified.  Otherwise, use the current system time.       *
 *************************************************************/

  /* Creation time */
  FIB_addr->FITCH.W =
      ((sys_pkt->CTIM == -1) ? system_time : user_time);

  FIB_addr->FITYP = sys_pkt->cftyp;
  FIB_addr->FIACL = 0;
  FIB_addr->FIUID = 0;
```

```
/*************************************************************
 * Nearing the end.                                          *
 * Fill in FNB pointer in the FIB, and FIB pointer in the FNB.*
 *************************************************************/

   FIB_addr->FINLP = convert_to_idp (FNB_addr);
   FNB_addr->FNFIB = convert_to_idp (FIB_addr);

/*************************************************************
 * Flush the completed FNB and FIB to disk.                  *
 * This causes the parent dir to be modified, so set the mod  *
 * bit in its FCB.  This will cause its Funny FIB to be      *
 * flushed when the file is closed.                          *
 *************************************************************/

   RELF (FNB_bh_addr);    /* Bye Bye FNB */
   RELF (FIB_bh_addr);    /* Bye Bye FIB */

   set_bit (PFCB_addr, BFBMD);

   if (DIRECTORY_TYPE_FILE)
      PFCB_addr->FBIDR += 1;      /* Count a new inferior dir. */

/*************************************************************
 * MOSTLY DONE!                                              *
 * The FAC pointer was initted to 0 without even a check for  *
 * ACL specification.  The DOACL.P variant of Access Control  *
 * Services is chained to.  The ISACL.P service could have    *
 * been used instead, but VS chose this approach.  An FAC will*
 * be allocated if more than a universal ACL is specified.    *
 *************************************************************/

   chain (DOACL.P);

}
```

2.4.3 File Deletion Services

File Deletion Services represents the functional opposite of File Creation Services. File deletion services deletes a file's FNB and FIB directory data elements in its parent directory. The FAC or FLB will be deleted as well. If a UDA was created for the file, the FUD will also be released. The REDEL operation will be called to accomplish standard DDE deletion.

File deletion encompasses more than interaction with directory data elements. Most files are associated with data elements, which must be deallocated from disk. File creation is not concerned with data element allocation; CCB Request Management allocates the data elements on I/O requests. File deletion, however, must take responsibility for deleting all of a file's data, including the index blocks that build the file's physical structure. Fortunately, the I/O world deallocates both data elements and index blocks upon request. File Deletion Services need only make one DELFIL ("DELete FILe") request to CCB Request Management to delete a file's data. Consequently, the file's first address will again be 0.

Although directories are files just like any other data file, their deletion is more complex. The contents of a directory are its DDEs, which hold information about the files residing in the directory, including their filenames. All DDEs plus the data housed by each file must be deleted. This implies that a simple ?DELETE could become a recursive operation traversing the directory tree from top to bottom. If this were allowed, a CLI DELETE (which does a ?DELETE) could wipe out entire directory hierarchies, a dangerous consequence for such a simple, common operation. AOS/VS stipulates that user directory deletes are possible only if none of the directory's subordinate files are directories. This forces only one recursive level within the DELETE operation: that of each of the mandatory non-directory files in the directory. (CLI parses the "#" template supplied in "DELETE" commands and subsequently makes the required number of ?DELETE system calls to service the user's request. "#" is not a ?DELETE parameter!) System initiated deletes do provide the option of relentless deletion of ALL files in an input directory.

Another interesting feature of Deletion Services is that if a file is open and a delete file request is made, only the file's FNB will be deleted, creating the illusion that the file is gone. In reality the FIB, FAC, FUD and all data elements remain allocated and intact in the directory until the file's FCB open count reaches zero (last close). This explains why the execution of a program file will not be aborted if the .PR file is "deleted" during execution. The file's physical structure and all its data is still accessible from disk by processes which already have the file open. Further opens of the file are not possible, since the filename no longer appears in the directory.

There are four variants of File Deletion Services. Each variant is accessed through a specific entry point. The different types of deletion services and a functional description of each follows:

Variant	Function
DELETE.P	Delete a file for a user. Implementation of ?DELETE system call.
IDELETE.P	Delete a file for the system. Internal file delete.
UDELETE.P	Delete a file for the system, on behalf of the user.
VDELETE.P	Delete an LDU entry from its parent dir.
CLDEL.P	Delete a file marked for "delete on last close."

External requesting modules of DELETE.P

Caller	Effect
User	?DELETE - Delete the file.
DOACL.P	Delete the file being created.

The user issues a ?DELETE, which passes through the system call processor and enters the system call code at DELETE.P. DOACL.P is a service chained to from CREATE.P (after the FNB and FIB have been created) to allocate the FAC and establish a file's ACL. If the ACL is invalid, the previously created DDEs must be deleted. This is done via a request to DELETE.P.

External requesting modules of IDELETE.P

Caller	Effect
BRKFL.P	Delete breakfile on error condition.
SDOWN.P	Delete :PER and :PROC.
SINIT1	Delete :PER and :PROC before recreation. Delete :SWAP and :PAGE before recreation. Delete invalid files in :SWAP and :PAGE
ALLSW.P	Delete a PID's swap file on ?PROC.
DALSW	Delete a PID's swap file on ?TERM.
PSWIPE.P	Delete all unopened swap and page files.

IDELETE.P is an internal ?DELETE. The necessary parameters are passed in the system accumulators rather than in the user packet. The system requests IDELETE.P service when it decides to delete a file. BRKFL.P utilizes IDELETE.P to delete the breakfile if an error condition arises after its creation. SDOWN.P calls IDELETE.P to delete the :PER and :PROC directories on system shutdown. System initialization makes the call as well in case system shutdown did not run to completion (or did not run at all). One option of the IDELETE.P call is to delete all files in a directory, whether or not they are directory files and regardless of access privileges and permanence. System initialization also calls IDELETE.P to delete files in :SWAP and :PAGE that are not in the correct format for the directory (e.g., SWAP.00099).

Process management (ALLSW.P) makes use of IDELETE.P by deleting a new PID's swap file on ?PROC if the file is hot (already exclusively open) and the new swapfile is a non-default size. DALSW calls IDELETE.P to delete a PID's swap file on ?TERM if it was created with a non-default size. Normally, swap files are left exclusively opened on process termination to avoid the overhead of closing on ?TERM and reopening on ?PROC. Finally, PSWIPE.P calls IDELETE.P for each unopened swapfile in :SWAP and pagefile in :PAGE when an error is returned from a swapfile I/O request. The error condition assumes that there is a potential fatal condition, since there is no reason for the I/O to fail. The call to PSWIPE.P attempts to free up disk space in :SWAP and :PAGE in case the I/O error was due to insufficient space, and the original I/O request is retried. If the second request fails, AOS/VS will panic with code 14413.

External requesting modules of UDELETE.P

Caller	Effect
BRKFL.P	Delete the existing filename with which a breakfile is to be created.

UDELETE.P is called only by BRKFL.P when a breakfile name is specified by the user in a ?BRKFL, ?MDUMP or ?TERM/BREAKFILE system call and the filename already exists. UDELETE.P is exactly the same as IDELETE.P, except that UDELETE.P will not delete the file if access is denied or permanence is set.

External requesting modules of VDELETE.P

Caller	Effect
DRLSE.P	Delete the LDU entry from its parent dir.

VDELETE.P is called to from LDU release code to delete the FNB/FIB/FAC from the parent directory.

External requesting modules of CLDEL.P

Caller	Effect
CLOSE.P	Delete the file just closed which was marked for "delete on last close."

CLOSE.P calls CLDEL.P to "really" delete the file when the FCB open count of a file marked for "delete on last close" reaches zero.

The input values to file deletion services are similar for all variants. An explanation of any variant algorithm should provide a clear understanding of the file deletion mechanism. The system internal delete service, IDELETE.P, is presented here in algorithm form.

Inputs and outputs of IDELETE.P:

Variable	Input	Output
AC0	Byte address of file pathname in sys space.	Unchanged.
AC1	Options flag.	Unchanged.
AC2	Not used.	Unchanged.

AC1 is zeroed if checking for permanence and access rights is to be done. Also, if the file to be deleted is a directory, and there are inferior directories as well, the directory will not be deleted. AC1 is loaded with one if all these checks are to be ignored.

```
LLEFB   0,PERNM*2,0     ;Byte pointer to ":PER"
NLDAI   1,1             :Ignore validity checks
LJSR    IDELETE.P       :Adios
  WBR   ERRTN
```

```
IDELETE.P (*pathname, options_flag)
{

/*************************************************************
 * Resolve pathname of file to delete.                       *
 * Returned are the FIB/FNB IDPs and parent dir CCB address.  *
 * If options_flag=0, make sure all access checking done.    *
 *************************************************************/

     aflag = SYSTEM_SPACE_BIT;
     if (options_flag == 0)
        options_flag = all_validity_checks_bits;

     call RESLV.P (pathname, input_switches, aflag,
                   &FIB_IDP, &FNB_IDP, &PCCB_addr, &dum1, &dum2)

/*************************************************************
 * If specified, check ACLs.                                 *
 * If caller has owner access to the file OR write access to  *
 * the file's parent directory, the file can be deleted.     *
 * Errors from ACL services returned in CERWD of the CB.     *
 *************************************************************/

     if (options_flag & check_ACL)
       {
       /* Check caller's access to the file to be deleted */
       call PESTAC.P (FIB_IDP, username, PCCB_addr, &ACL_privs);

       if (!(ACL_privs & owner_access))
         {
         /* Check caller's access to the directory */
         call ESTAC.P (PCCB_addr, &ACL_privs);

         if (!(ACL_privs & write_access))
            {
            return (error);
            }
         }

/*************************************************************
 * Access allowed; read in the file's FNB and FIB.           *
 *************************************************************/

     call RAID (FIB_IDP, DEFIB, PCCB_addr, &FIB_bh_addr);
     call RAID (FNB_IDP, DEFNB, PCCB_addr, &FNB_bh_addr);

/*************************************************************
 * If specified, check permanence in FIB.                    *
 * Delete the file, even if permanent, if caller so specified.*
 *************************************************************/

     if (options_flag & check_permanence)
       {
       if (FIB_addr->FISTS & permanence_bit)
         return (cannot_delete_permanent_file);
       }
```

```
/************************************************************
 * Check if the directory and all its files should be deleted.*
 * If there is no FCB, the file is closed and can be deleted. *
 * If there is an FCB (file open), no inferior dirs and the   *
 * system CCB use count is 0, it will be deleted.             *
 *                                                            *
 * Notice that directory deletion will be attempted if        *
 * the delete_ALL_files_in_dir option is selected, even if the*
 * dir is open (it will be marked for delete on last close    *
 * later).  Decrement the count of inferior dirs in the      *
 * parent CCB now.                                            *
 ************************************************************/

    if (FIB_addr->FITYP == directory_type)
      {
      if (!(options_flag & delete_ALL_files_in_dir))
        {
        if (FIB_addr->FIFCB.W != 0)
           if (FIB_addr->FIFCB.W->FBIDR != 0)
              return (Directory_delete_error)
           else if (FIB_addr->FIFCB.W->FBSCB.W->CBUSC != 0)
              return (Directory_in_use_error);
        }

      PCCB_addr->CBFCB.W->FBIDR -= 1;
      }

    /* Must ?RELEASE LDU type directories (VDELETE.P) */
    else if (FIB_addr->FITYP == ?FLDU)
      return (Directory_delete_error);


/************************************************************
 * Delete checks out.  Delete the file's FNB!                 *
 ************************************************************/

    call REDEL (PCCB_addr, FNB_bh_addr, FNB_addr);
    FIB_addr->FINLP = 0;

/************************************************************
 * If the file is open, mark if for delete on last close.     *
 * The file will be deleted (CLDEL.P) when the FCB open count *
 * reaches 0.                                                 *
 ************************************************************/

    if (FCB_addr->FBOPN > 0)
      {
      set_bit (FCB_addr, BFDBL);
      RELM (FIB_bh_addr);
      return;
      }
```

```
/*************************************************************
 * If the file to delete is a dir, delete the files in it.    *
 * File Deletion Services basically implements this as a      *
 * recursive IDELETE.P for all the files.                     *
 * See the code for more details (AOS/VS module: DE2.SR)      *
 *************************************************************/

     if (FIB_addr->FITYP == directory_type)
         {
         for (each file in the directory)
            call IDELETE.P (filename, options_flag=1);
         }

/*************************************************************
 * Enqueue the request to delete the file's data elements.    *
 * This is done by putting a "DELETE FILE" command in the CCB *
 * flag word and "enqueuing the CCB."  See Section 3.3.       *
 * The file will be null after this call and its DDEs in the  *
 * parent directory can be released!                          *
 *************************************************************/

     PCCB_addr->CBFLG = CBDEL;
     call NQCCB (PCCB_addr);              /* Delete file data */

     if (FUD_IPD = FIB_addr->FIFUD)       /* Delete FUD       */
         {
         call RAID (FUD_IDP, DEFUD, PCCB_addr, &FUD_bh_addr);
         call REDEL (PCCB_addr, FUD_bh_addr, FUD_addr);
         }

     if (FIB_addr->FITYP == ?FLNK)        /* Delete FLB       */
        {
        if (FLB_IDP = FIB_addr->FINLP)
           {
           call RAID (FLB_IDP, DEFLB, PCCB_addr, &FLB_bh_addr);
           call REDEL (PCCB_addr, FLB_bh_addr, FLB_addr);
           }
        }
     else
        if (FAC_IDP = FIB_addr->FIACL)    /* Delete FAC       */
           {
           call RAID (FAC_IDP, DEFAC, PCCB_addr, &FAC_bh_addr);
           call REDEL (PCCB_addr, FAC_bh_addr, FAC_addr);
           }

     if (FIB_IDP = FIB_addr->FIACL)       /* Delete FIB       */
        {
        call RAID (FIB_IDP, DEFIB, PCCB_addr, &FIB_bh_addr);
        call REDEL (PCCB_addr, FIB_bh_addr, FIB_addr);
        }

/*************************************************************
 * DONE!                                                      *
 *************************************************************/

     return;

}
```

3 File Management

3.1 Overview

File Management provides both the user and system components with an interface to lower level I/O. There are two primary functions of File Management services:

1) to build and maintain file-specific databases,
2) to process and initiate user and system I/O requests.

The two main file-specific databases are the File Control Block (FCB) and the Channel Control Block (CCB). These databases hold dynamic information relating to the state of an open file and to the state of the current I/O request on that file, respectively. File Management is responsible for allocating, initializing, and releasing these databases when files are opened and closed.

The mere existence of these databases, which correspond to a unique file, advises AOS/VS that certain sections of disk-based data are vulnerable to modification (files are open). When a process is terminated, its open files must be closed and modified files, whose buffers may still be in memory, must be flushed to disk. The user CCBs and the files' FCBs must be retrieved and possibly deallocated. Likewise, when AOS/VS shuts down (either normally or after a panic), modified buffers must be flushed to disk and open files must be closed. The system shutdown routine prints out the occurrence of these procedures on the master console ("Flushing buffers" and "Open file processing").

File Management services further initialize the databases with I/O-related parameters before making an I/O request. After a file is opened, either for read only or write access, some type of I/O is usually requested. The FCB and CCB (mostly the latter) provide the specific data necessary to run an I/O request. Most "customers" of File Management services run on control blocks, because they run the risk of pending awaiting I/O completion. However, some services do not pend even though an I/O request is enqueued. Each particular service is designed to manage its callers' control blocks and process tables in either case.

There are few direct user interfaces to file management services. They are:

1) file opens, e.g., ?OPEN, ?GOPEN, ?SOPEN
2) file closes, e.g., ?CLOSE, ?GCLOSE, ?SCLOSE
3) file I/O requests, e.g., ?RDB, ?WRB, ?PRDB, ?PWRB, ?BLKIO

The system has access to these as well as I/O initiation services, which are invoked either by the authoritative decision of another system component or by a file system component on behalf of the user. For example, NQCCB is a service that creates, initializes and schedules an I/O request, and alerts the disk manager to run the request on behalf of a user process. The RDB.P (implements ?RDB) service calls NQCCB. NQCRQ is a generic ring 0 interface service that implements a similar procedure; the request is made by the system without explicit user direction. The RDUST.P service (reads .PR file UST) calls NQCRQ.

File management services can be learned by understanding the purpose and contents of the most important databases. Once this knowledge is incorporated, comprehension of the specific services will follow. This chapter introduces high-level, File Management concepts and provides excellent insight into lower-level CCB Request and Buffer Management services.

3.2 File Control Block (FCB)

3.2.1 FCB Parameter Definitions

As explained in Section 2.2, system main memory is allocated for FCB creation and destruction. An FCB is created when a file is opened for the first time and destroyed when the file is closed for the last time. There is only one FCB for any given open file, whether opened by the user or by the system. Hence, the FCB contains common data regardless of how the file is opened. Some parameters remain static throughout the life of the FCB, such as the file's parent directory CCB pointer and its depth in the directory hierarchy, indicating fixed properties of the file. Most parameters are dynamic, such as the file open count and file status word, monitoring the changing state of the file. The following diagram briefly describes each of the FCB parameters.

File Control Block (FCB)

Offset		
FBLCB.W	0	Logical Unit Control Block (LCB) address.
FBBLP.W	2	FCB Buffer List Queue Descriptor Pointer (head).
	4	FCB Buffer List Queue Descriptor (tail).
		Here starts the FUNNY FIB.
FBSTS	6	Status (Bits 0-10), Universal ACL (Bits 11-15).
FBTYP	7	File Format (Bits 0-7), File Type (Bits 8-15).
FBHFS	10	Hash Frame Size: if directory type file
FBCPS	10	File Control Parameters: Index if generic file. Record length if file open for fixed length I/O.
FBDCU	10	Device Code (Bits 0-7), Unit number (Bits 8-15).
FBFW1	11	Extension for EOF in future. (Now 32-bit max)
FBEFH.W	13	Number of bytes in file (byte EOF).
FBDFH.W	15	Data element size. Set at file creation.
FBFAH.W	17	File First Logical Address. Zero if null file.
FBIDX	21	Current (Bits 0-7), max (Bits 8-15) index levels.
FBIDR	22	Count of inferior directories (dir type files).
		This is the end of the Funny FIB.
FBOPN	23	File Open Count.
FBUID	24	FCB Unique ID.
FBPDP.W	25	Parent Directory CCB Pointer.
FBFIB	27	FIB Intra-Directory Pointer (IDP).
FBCMP.W	30	FCB Shared Page Header Queue Descriptor.
FBSCB.W	34	System CCB Pointer.
FBUNDC	36	Unit number (Bits 0-7), device code (Bits 8-15).
FBLOCK	37	FCB Lock Word.
FBWTC	40	IOCB waiter count on this FCB.
FBLVL	41	Depth of file in directory hierarchy.
FBCPB.W	42	Control Point Block (CPB) pointer.
FBUDB.W	44	UDB address if the file Unicorn LPU or MTU type.
FBST2	46	Shared protected file status word.
FBFOP	47	Pid and Ring of first opener of ?SOPPFed file.
FBPPB.W	50	PPB chain queue descriptor (head).
FBPPBB.W	52	PPB chain queue descriptor (tail).
FBLMB.W	54	Lock Management Block pointer.
FCBLT	56	Length of FCB.

The LCB address of the LDU on which the file resides must be stored in the FCB. CCB Request Management determines the correct logical unit to enqueue requests by examining this field. The LCB points to the unit definition block (UDB) list, which describes each of the units in the LDU. Ultimately, the disk driver will enqueue the request to the proper physical unit.

Buffer headers found on the FCB buffer list queue (FBBLP.W) represent directory data blocks, directory bit map blocks, file blocks from system-initiated I/O (IPC spoolfile, HIF file and PIF file blocks), or all file index blocks. When CCB Request Management reads in a single data block ("read system buffer" CCB command) or an index block (during index level traversal) into a system buffer, the buffer header is enqueued to the FCB. The logical disk address of the blocks are stored in the buffer header. The relative block number is stored in the buffer header for "read system buffer" data blocks, but the same field (BQDBN) is filled with -1 for index blocks to distinguish them from data blocks. Data blocks read or written during all other CCB command processing (all user I/O) are not enqueued to the FCB buffer list because the buffer header is not a system buffer header (see Section 4.2.7).

Before file blocks are read from disk, the FCB buffer list queue is searched for a match on the desired logical disk address. If the block is found, no disk I/O request need be made. This minimizes the number of enqueued disk requests and interrupt service time, and improves general system performance. When the last close on the file occurs, the buffer headers are moved to the LCB cache buffer list queue. When the file is deleted, the buffer headers are removed from either the FCB or LCB buffer lists. See CCB Request Management and Buffer Management for more specific details.

```
FBBLP.W ---> BH <===> BH <===> BH <===> BH <===> BH <===> -1
```

The Funny FIB on disk is copied into the FCB Funny FIB common area on the first file open. Throughout the open life of the file, the Funny FIB parameters are maintained in the FCB. This is a logically comprehensible operation. If this implementation were not chosen, the FIB would either have to be maintained somewhere else in main memory while the file was open (probably with a word pointer in the FCB), or it would have to be read/modified/written to disk upon any change. The latter approach would not be feasible since the extra disk I/O overhead is not necessary. The former approach has been implemented, in effect, by reading the most dynamic portion of the FIB directly into the FCB.

The file open count (FBOPN) is a dynamic parameter initialized to 1 on the first file open. File Open Services increments the count for each successive open on the file. When the count reaches 0, the FCB buffers as well as the Funny FIB portion of the FCB will be flushed to disk and the FCB will be destroyed.

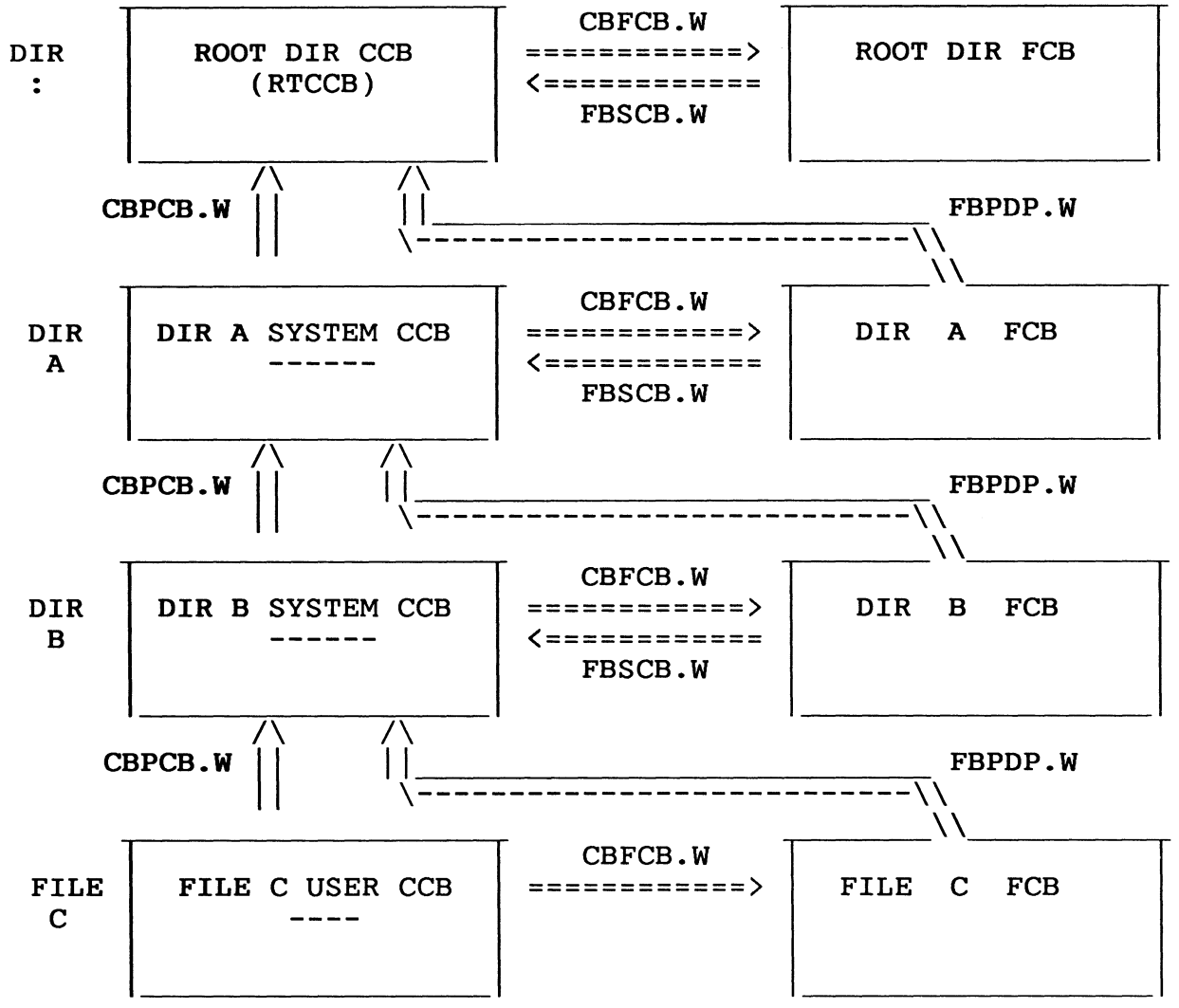
The FCB unique ID (FBUID) is a static parameter initialized to the value of the AOS/VS global variable IFCB. IFCB is initialized to 1 and is incremented each time the value is assigned to a new FCB. The purpose of FBUID is to verify the FCB/CCB association of the same file. The CCB unique ID (CBUID) is assigned the value of the file's FBUID. When a channel request is made, the file's FCB and CCB must be retrieved. Once they are both in the hands of the caller, a sanity check is made on their unique IDs. If they do not match, AOS/VS will panic with code 6034.

The parent directory of any file is the directory in which the file resides. The parent directory of :UTIL is the root (:). The parent directory CCB address of an open file is stored in FBPDP.W. Since the file's FNB, FIB and FAC are situated in the directory data blocks of its parent directory, this field is essential for I/O to be accomplished. Furthermore, the file's FIB pointer (IDP) is found in the FCB as well (FBFIB). As a result, Resolution Services return both the parent directory CCB address and the FIB pointer of the filename being resolved.

The master root is the only "orphan" directory. The parent directory fields in the root FCB and CCB are zeroed. Certain operations are illegal on the root directory, such as deleting it! The root directory FCB and CCB are identified by this zero value, and a simple check in DELETE.P signals that someone is attempting to delete the root. Moreover, any operation that requires modifying the parent directory, such as set ACL or RENAME, is illegal for the root. Since the root directory's FIB type information lives in the DIB, such operations are allowed only by the DFMTR utility program.

If a file is being opened by Resolution Services, both an FCB and a system CCB must be created. The FCB is created first, followed by the system CCB. The system CCB address of the same file is stored in the FCB at offset FBSCB.W. When this happens, the file open count (FBOPN) is not incremented, but the indication that the file is "implicitly" open is through the non-zero value at FBSCB.W. If the file is being opened by the system via some internal open call (IOPEN.P, XEOPEN.P, etc.), a system CCB is allocated from GSMEM, but FBSCB.W is not filled in, and FBOPN is incremented. See CCB Creation/Destruction for more details.

The parent/son/system CCB structure links together the FCB/CCB chain, from the root directory down through the hierarchy to the last file opened in the pathname. This enables Close Services to systematically access and destroy all system FCBs/CCBs that were created in order to open the file. Resolution Services establishes this chain when opening the files in a given pathname on system FCBs/CCBs. When the system explicitly opens a file on a system CCB, it will be the lowest CCB in the chain. The following diagram outlines the open FCB/CCB chain that exists for file :A:B:C when file C is opened by a user.



When shared file pages are read into a process' working set, a shared page header (SPH) is allocated and enqueued to the FCB Shared Page Header Queue whose queue descriptor is at offset FBCMP.W. The SPH holds data associating the shared memory page with its logical disk location. The FCB SPH queue is traversed when an operation on all the file's SPHs must be done, such as ?ESFF or ?SCLOSE. The SPH is linked to the appropriate system shared page hash chain as well. The system SPH hash chains are searched for specific SPH access. (Refer to Shared Memory Management for more on SPHs.)

A new feature of AOS/VS 7.00 is File Locking. File locking provides a mechanism in which cooperating processes are able to communicate by pre-defining a series of file elements and requesting either exclusive or shared access to these elements. Exclusive file locking can be used to prevent cooperating processes from accessing the same object at the same time (mutual exclusion within a critical region). Shared file locking can be used to give several processes simultaneous access to an object, but prevent another process exclusive

access. Users access file locking features via the ?FLOCK/?FUNLOCK system calls. When file locking is activated on a file, a Lock Management Block (LMB) is created, and its address is stored in the FCB at offset FBLMB.W.

The FCB has a lock word (FBLOCK) in which three lock bits are defined. The lock bits are:

Bit position in FBLOCK	Bit position in FCB	Function
FBTRAN (0)	BFBTRAN	FCB transition lock
FBCMPLK (1)	BFBCMPLK	FCB SPH Queue lock
FBLMBB (2)	BFBLIP	FCB LMB in use lock

The FCB transition lock is acquired when the FCB buffer list is to be accessed, e.g., when a buffer header must be enqueued or dequeued. The FCB transition lock has been implemented in AOS/VS 7.50 due to multiprocessor considerations. In previous revisions there was never a danger of the FCB buffer list queue being accessed by simultaneously executing code paths. The multiprocessor environment forces queues to be locked when elements are searched, enqueued, dequeued or modified in order to preserve the integrity of the linked list. By definition, transition locks are "short-term locks" and can therefore be implemented as spin locks.

The SPH Queue lock is acquired when the FCB SPH Queue is to be accessed. The LMB in use lock is acquired when any ?FLOCK/?FUNLOCK operation is in progress within the operating system. Each of these locks is a spin lock, acquired by calling one of the standard AOS/VS spin lock routines. For example, with the following assembly code sequence:

```
NLDAI   BFBTRAN,1       ;Lock bit offset from
XWLDA   2,FCB_addr,3    ;  the FCB.
XPSHJ   FXLOCK          ;Get the spin lock.
```

The IOCB waiter count (FBWTC) is initialized to 0 and incremented by CCB Request Management when the I/O in progress bit (FBIOP) is already set and an IOCB is enqueued with another disk request on the same file. When IOCB requests complete, this field is checked for a non-zero value. If it is non-zero, the next waiting IOCB ready to run will be readied and FBWTC is decremented by 1. (There can be multiple waiters!)

The depth of a file in the directory hierarchy, relative to the root LDU on which the file resides, is maintained in bits 8-15 of FCB_addr->FBLVL. This is called the file's "local" level. The deepest local level at which a file can exist is 8 (system parameter SCLVL). Hence, file :1:2:3:4:5:6:7:8:9 is an impossibility if none of the subordinate files of the pathname are LDUs. The depth of a file in the directory hierarchy, relative to the system root LDU, is maintained in bits 0-7 of FCB_addr->FBLVL. This is called the file's "global" level. The deepest global level is 255. This parameter is set by File Open Services or LDU initialization.

The control point block (CPB) address is stored in the FCB at offset FBCPB.W. The CPB contains parameters indicating the CPD's current and maximum space availability. System memory is allocated for a CPB when a control point directory (CPD) is opened, and maintained in memory until the file is closed for the last time. The function of the CPB is analogous to that of the Funny FIB (in the FCB). The CPB is described in Directory Management Databases, Section 2.2.

Certain unit files require the presence of a Unit Definition Block (UDB) to realize unit I/O. Unicorn type line printers and magnetic tape units are such files. The UDB address is stored in the unit file's FCB at offset FBUDB.W. Non-Unicorn line printers and MCAs require only a buffer header, which is stored in its CCB (CBBHR.W). Disks opened as unit files also require a UDB, but it is linked to the LCB. The device code and unit number of all unit files are stored in the FCB at offset FBDCU.

The FCB maintains shared protected file information. When a file is opened for shared protective access (?SOPPF), bit 0 (FBPFO) is set in the second FCB status word, FBST2. Since no subsequent openers are permitted more access rights to the file than the first opener, the first opener's access privileges are stored in bits 11-15 of FBST2. Subsequent openers of shared protective files must be customers of the first opener, if they wish to open the file. They must explicitly be granted permission to access the file as well. The ?PMTPF system call enables the first opener to specify distinct, privileged PIDs permitted to open the file, as well as the access rights to be granted. The file access information established by the first file opener for another PID is stored in a separate Protected File Permission Block (PPB), which is linked to the FCB through the queue descriptor found at offset FBPPB.W. Only the first file opener can successfully issue ?PMTPF against its customers. ?PMTPF prevents unauthorized callers from successfully creating PPBs by comparing the caller's PID and ring with the PID and ring of the first opener, stored in the FCB at offset FBFOP when the file was ?SOPPFed. There is no special FCB lock bit for FBPPB.W because the parent CCB remains locked during all queue instructions.

There are several file types for which FCBs are not created even though they are open. Generic files (?FGFN) are resolved in the AGENT. Opening @LIST actually results in opening its resolution filename, thus creating an FCB for the latter. IPC file (?LIPC-?HIPC) FIBs contain the global port number associated with the IPC file. IPC file opens consist of user CCB creation and user retrieval of the global port number. No FCB is needed for IPC Management to send and receive messages. EXEC Queue files (?FQUE) files do not need FCBs either because they are handled by the AGENT.

3.2.2 FCB Creation/Destruction

Since the FCB length is only 46 words, File Management does not allocate small chunks of random system memory for each individual FCB. Instead, whole system pages are dynamically allocated and reserved for the exclusive use of FCBs. These pages are called FCB pages. The total number of FCBs that fit on one page is easily calculated:

PAGESIZE/FBBLT = 1024./46. = 22.

Associated with each FCB page is an FCB page descriptor. The FCB page descriptors are linked through the global queue descriptor FCBCH.W and describe the contents of the FCB page. These descriptors provide a quick mechanism for Emergency Shutdown (ESD) to locate open files, all of which must be closed. The following diagram illustrates the FCB page descriptor:

Offset		FCB Page Descriptor
CMSFL.W	0	Forward link.
CMSBL.W	2	Backward link.
CMFBK	4	Physical page number of FCB page.
CMFCN	5	Number of FCBs in use on this FCB page.
CMBMW.W	6	FCB page bit map word.
CMFLN	10	Length of FCB Page Descriptor.

When an FCB needs to be created, offset CMBMW.W of the first FCB Page Descriptor is examined. Each bit in CMBMW.W corresponds to the FCB whose offset into the FCB page is the bit position multiplied by the FCB length. A set bit (1) in CMBMW.W represents a free FCB. Since the maximum number of FCBs that fit on the page is 22, any set bit between 0 and 21 indicates a free FCB. If there are no free FCBs (no FCB Page Descriptors left), a page along with an FCB Page Descriptor will be allocated from general system memory. The FCB Page Descriptor will be initialized and enqueued to FCBCH.W, and FCB memory will then be available.

The following global variables used by File Management are associated with FCBs:

Global	Function
FCBCH.W	FCB Chain Queue Descriptor
FCBCN	Number of FCBs currently in use
FCBMX	Max number of FCBs in use since boot
IFCB	FCB Unique ID counter

3.2.3 FCB Operations: Get File Control Block (GFCB)

GFCB performs the operation of searching FCBCH.W and finding a free FCB. There are no inputs to GFCB. The only output, returned in AC2, is the new FCB address. There are only two callers of this operation:

Caller	Function
FFCB	If the file is open, returns the FCB addr (from FIFCB.W in FIB) to the caller. If file is not open, calls GFCB to create the FCB and then initialize its fields.
LNKLCB.P	Calls GFCB to create an FCB for an LDU being initialized (among other things).

FFCB is called by DELETE.P to create an FCB when deleting a file that was not open, by GOPEN.P to create an FCB when a file is being opened, and by RESLV.P to create FCBs (associated with system CCBs) when resolving pathnames.

3.2.4 FCB Operations: Release File Control Block (RFCB)

RFCB performs the operation of destroying FCBs and returning them to the pool of free FCBs on the FCB page. The corresponding bit in the bit map word of the FCB Page Descriptor will be cleared. If the last FCB on the page is freed, the FCB page will be deallocated and returned to system memory. The only input to RFCB is the FCB address in AC2. The callers of RFCB are the following:

Caller	Function
FFCB	On error, must destroy the FCB it created.
XINIT.P	On error, must destroy the FCB it created.
DRLSE.P	Destroys FCB on LDU release.
CLOSE.P	Destroys FCB on last file close.
DELETE.P	Destroys the FCB it created to delete a file that was not previously open.

3.2.5 FCB Operations: Kill File Control Block (KFCB.P)

KFCB.P is the operation called after user CCBs are destroyed, i.e., the file is being closed. Therefore, the FCB open count must be decremented. If the open count reaches zero, KFCB.P releases all memory resources tied in with the FCB, including system buffers and allocated memory for databases, and finally destroys the FCB. System CCBs are destroyed by KCCB.P, which is followed by execution of KFCB.P code. AOS/VS is written in assembly language, in which it is easy to LJMP from CCB specific code to FCB specific code and vice versa. The implementation of KFCB.P and KCCB.P contains overlapping code accessed via LJMPs. This algorithmic code cannot do the same, because so much of the KFCB.P code is duplicated in KCCB.P. The concept of FCB/CCB destruction will be clear.

The inputs to KFCB.P include the MCA link number and DCT (if MCA being closed) and the FCB address (always). The callers of KFCB.P are the following:

Caller	Function
GOPEN.P	On error, must destroy the FCB it created.
SOPPF.P	On error, must destroy the FCB it created.
UNIT.P	On error, must destroy the FCB it created.
ESFCB	Destroys FCBs of open files on ESD.
CLOSE.P	Destroys FCB of file on last close.

```
KFCB.P (MCA_link_num, MCA_DCT_addr, FCB_addr);
{
/*************************************************************
 * Write an EOF to the MCA device if the MCA is being closed. *
 *************************************************************/

   if (FCB_addr->FBTYP == ?FMCA)
      call MCACLS (MCA_link_num, MCA_DCT_addr);


/*************************************************************
 * KFCB.P is called right after the user CCB was destroyed.   *
 * This routine assumes that the child CCB was just released, *
 * but RUCCB.P does not decrement the CBUSC.  It is done here *
 * because sometimes the caller of RUCCB.P will not want it   *
 * done (when called on error return BEFORE parent CCB        *
 * stored).                                                   *
 *************************************************************/

   PCCB_addr = FCB_addr->FBPDP.W;
   if (PCCB_addr)
      {
      call CLOCK (CCB_addr);              /* Lock parent CCB */
      PCCB_addr->CBUSC -= 1;              /* One less CCB son*/
      }
```

```
/*************************************************************
 * If the file was opened on ?SOPPF and is now being closed  *
 * by the first opener, tear down the PPBs on the FCB now.   *
 *************************************************************/

   if (get_bit (FCB_addr, BFBPF)) && first_opener)
      {
      FCB_addr->FBFOP = 0;
      call PCLOCK;                              /* Lock CNXTB */
      call TDPPBL.P;                    /* Tear down PPB list */
      call PCUNLCK;                          /* Unlock CNXTB */
      }

/*************************************************************
 * File being closed; decrement file open count.             *
 * If file still open, check if the Funny FIB must be flushed.*
 * This part of the code is only called when a user CCB was  *
 * released.  If the file is a directory, users only have    *
 * read access, and the Funny FIB will not have changed.     *
 * In all other cases it may have, so release the buffer     *
 * modified.  The file is still open, so the data need not be *
 * actually flushed now.  It will be flushed either at last  *
 * close or when someone attempts to assign this modified    *
 * system buffer.  Then unlock parent CCB and return.  (CCB  *
 * gone, FCB not)                                            *
 *************************************************************/

   if (--FCB_addr->FBOPN != 0)          /* Is file still open? */
      {
      if (FCB_addr->FBTYP != DIRECTORY_TYPE_FILE)      /* YES */
         {
         call RAID (FCB_addr->FBFIB, DEFIB, PCCB_addr,
                    &FIB_BH, &FIB_addr);
                                                   /* Read FIB */
         call UPFIB (FIB_addr, FCB_addr);         /* Update it */
         call RELM (FIB_BH);                /* Will be flushed */
         }

      if (PCCB_addr != 0)
         call CULCK (PCCB_addr);

      return;
      }

/*************************************************************
 * Last opener is closing file.  Clean up the FCB.           *
 * RELBF: Release buffers on FCB buffer list and enqueue them *
 *        to the LCB cache buffer list.                      *
 * RELSP.P: Release shared pages on FCB.                     *
 *************************************************************/

   call RELBF (FCB_addr);               /* Release FCB buffers */
   if (FCB_addr->FBCMP.W != -1)         /* and shared pages if */
      call RELSP.P (FCB_addr);          /* there were any.     */
```

```
/*************************************************************
 * If the file was marked for delete on last close, delete it *
 * now!  Then jump to KCCB to release the system CCB/FCB      *
 * hierarchy above.                                           *
 *************************************************************/

   if (get_bit(FCB_addr->BFBDL))
      {
      call CLDEL.P (FCB_addr->FBFIB, PCCB_addr, FCB_addr);
      goto KCCB.P (PCCB_addr);
      }

   /* If unit type file, do unit independent operations */
   if (FCB_addr->FBTYP == UNIT_TYPE_FILE)
      call UCLOSE.P;

/*************************************************************
 * Read in the FIB, update it with the modified data from the *
 * FCB's Funny FIB and flush it out to disk NOW.  Flush now   *
 * (RELF) because no other users have file open.              *
 *************************************************************/

   call RAID (FCB_addr->FBFIB, DEFIB, PCCB_addr,
              &FIB_BH, &FIB_addr);

   FIB_addr->FIFCB = 0;
   call UPFIB (FIB_addr, FCB_addr);
   call RELF (FIB_BH);

/*************************************************************
 * Almost done.                                               *
 * Release CPB memory.  Destroy the FCB!                      *
 * Then decrement the parent dir CCB use count.  If there are *
 * still users, just unlock it and return.  If it becomes 0,  *
 * it can be released (by KCCB.P).                            *
 *************************************************************/

   call RSMEM (FCB_addr->FBCPB.W, CPBLT);  /* Release CPB mem */
   call RFCB (FCB_addr);                    /* Annihilate FCB  */

   if (--PCCB_addr->CBUSC)                 /* One less FCB son */
      {                                    /* PCCB use count 0?*/
      call CULCK (PCCB_addr);              /* No, unlock it,   */
      return;                              /* leave hierarchy. */
      }

   goto KCCB.P;                            /* Yes, release it  */

} /* end of KFCB.P */
```

3.3 Channel Control Block (CCB)

3.3.1 CCB Requests

There is a unique CCB created for each file opener. If AOS/VS opens the file, a CCB is allocated from general system main memory. The system can open a file in two ways:

1) implicitly, in resolution services while resolving a pathname, or
2) explicitly, when making special internal open calls, such as for per process page, swap and IPC spool files.

A CCB created by an implicit open is called a "system CCB." A CCB created by an explicit open is called an "internal CCB." The differences between them are discussed more thoroughly in CCB Creation/Destruction.

If a user opens the file, a "user CCB" is created in ring 1 of the process' logical address space. Since multiple processes can have the same file opened simultaneously and request different size data transfers from different points in the file, each CCB serves the purpose of storing the necessary data to service individual I/O requests.

Since all disk I/O requests involve driver intervention with the physical disk controller and interrupt service, requestors must pend. The CCB contains a pointer to the caller's process table and TCB so that the correct task can be unpended upon data transfer completion. The actual procedure that AOS/VS follows when initiating a logical disk I/O request includes "enqueuing a CCB request." This terminology implies that the CCB is the database representative of unique I/O request data. Often in AOS/VS code and throughout this manual, "enqueuing a CCB" will be substituted as an understood abbreviation for "enqueuing a CCB request."

There are six types of CCB requests. Each one can be made either on behalf of the user (system call interface) or by choice of the operation system (i.e., read in swapfile, read a FIB). File Management provides services that implement the system calls to issue CCB requests on behalf of the user. File Management also provides services that issue CCB requests for other components of the operating system. The following table lists the available CCB request types, and the request type stored into the CCB command word before the request is enqueued.

Request Type	CCB Command	Function
Read	CBRED (0)	Vanilla read from a file
Read Sys Buffer	CBSYB (3)	Read one file block into a system buffer
Write	CBWRI (1)	Vanilla write to a file
Allocate	CBALL (5)	Allocate file blocks
Delete File	CBDEL (2)	Delete a file
Truncate File	CBTRN1 (4) CBTRN2 (6)	Truncate a file to a specified byte EOF (implemented in 2 parts)

3.3.2 CCB Parameters Definitions

The following diagram summarizes the significance of each field in the CCB.

```
Offset                    Channel Control Block (CCB)
------
CBQLK.W  0  Global Wait List Queue (CCBWQ.W) forward link.
CBFCB.W  2  FCB address.  See Section 3.2.1.
CBNBK    4  Number of blocks to transfer.

CBPTA.W  5  Process Table or Control Block address.

CBIAH.W  7  These three offsets are used by
CBXIA.W 11  CCB Request Management for file
CBIBN   13  index level traversal.

CBPCB.W 14  Parent directory CCB pointer.
CBFIB   16  FIB IDP (in parent directory).
CBDBH.W 17  Last block byte count (Bits 0-8).
            Relative disk block number in file (Bits 9-31).

CBUAD.W 21  User data buffer logical address.
CBSTS   23  CCB Status Word, File ACL, Unit Type.
CBFLG   24  CCB Flag and Command Word.

CBTCB.W 25  User TCB or Control Block address (to unpend).

CBUID   27  CCB unique ID.
CBUPD.W 30  CCB Post Processor address.
CBNPG   32  Number of referenced user pages in user buffer.
CBUSC   32  CCB Use Count: redefinition for dir type files!
CBPRI   33  CCB request priority.  Equal to process PNQF.

CBFAB.W 34  First Allocated Block read after "hole" in file.
            (For Read Next Allocated Elem option with ?BLKIO)
CBOIBN  36  Not used.

CBLOCK  37  CCB lock word.

CCBLT   40 Length of CCB.
```

In order for CCB Request Management to service a logical disk I/O request (also referred to as a CCB request), an I/O Control Block (IOCB) must be allocated from the IOCB.DB database pool. If no IOCB is available, the requesting CCB is enqueued to the global CCB wait queue CCBWQ.W in priority (PNQF) order. The CCB is linked through offset CBQLK.W. When a request completes and its IOCB becomes free, the first waiting (i.e., highest priority) CCB is dequeued and assigned the available IOCB for the I/O request to begin. Global variable CCWC holds a count of the number of CCBs currently on CCBWQ.W, and CCMX holds the maximum number of CCBs on CCBWQ.W since system boot.

CCB Request Management schedules I/O requests on the basis of the caller's priority enqueue factor (PNQF) found in the caller's process table. Hence, the caller's PNQF is stored in the CCB at offset CBPRI. The caller's process table is initially located by the I/O requestor through the active control block at CC.W. Since the main objective of CCB post processing is to unpend task control blocks (TCBs) or control blocks (CBs) that are pended awaiting I/O completion, the requestor's process table address is stored in the CCB at offset CBPTA.W. The TCB address is stored at CCB offset (CBTCB.W) if the very last operation to be performed by the requesting code path is that of enqueuing the request. For example, once RDB.P and WRB.P (?RDB/?WRB implementation code) initialize the CCB parameters and enqueue the I/O request, there will be no more system processing to be done upon I/O completion. In that case, the control block can be released (system call effectively completes), but the calling user TCB remains pended until the I/O completes. The CB address is stored at the same CCB offset if the requesting path must return to system code to complete processing (all other cases). This is illustrated in the CCB Read/Write request algorithms in Section 3.4.5.

The routine that unpends CBs and TCBs awaiting I/O completion is called the CCB Post Processor. The Post Processor is saved in the CCB (offset CBUPD.W) by the requesting service before enqueuing the CCB to the I/O world. It is called from CCB Request Management when the I/O request is complete. There are five flavors of CCB Post Processors because different requests require slightly different operations. This is clearly exemplified by the fact that a TCB is pended for user read/write requests, while a CB is pended for user delete requests. Read system buffer requests necessitate the transfer of the system data address to the caller, a special feature accomplished in the CCB Post Processor. A ?BLKIO request may specify the Read Next Allocated element option for which yet another CCB Post Processor is designated. All versions return any required data to the user packet. The following table outlines the various CCB Post Processor routines. See Section 4.5 for more detail.

CCB Post Processor	Function
PPCUS	Post Processor for user read/write/allocate and physical read/write requests.
PPCUB	Post Processor for user ?BLKIO requests.
PPCSY	Post Processor for system read/write, all delete/truncate requests.
PPCSR	Post Processor for shared read requests.
PPCBI	Post Processor for the "Read System Buffer" request. Issued only by AOS/VS.

The caller of CCB requests must initialize certain other CCB parameters with the request specifications. If the caller is a user (making a system call), these specifications are retrieved by AOS/VS from the user packet. Read, write and allocate block requests must set the number of blocks to be transferred in CBNBK. CCB Request Management decrements this count by n each time it enqueues a buffer header to transfer n number of blocks. When the I/O request is complete, this field is null. Delete and truncate requests set CBNBK to 0, since these operations do not require use of this field.

Offset CBDBH.W is composed of two separate fields. Bits 0-8 hold the last block byte count. On write operations, this field specifies the number of bytes in the last data block to transfer. For example, to transfer 768 bytes of data, CBNBK is set to 2 and the last block byte count to 256. On a read operation, CCB Request Management fills in this field when the data transfer completes. Bits 9-31 of CBDBH.W specify the logical data block in the file where the transfer begins. Delete requests do not actually involve disk-to-user data transfers, but file deletion begins from block 0. This field contains 0 if the CCB command is CBDEL. Truncate requests work similarly.

Offset CBUAD.W holds the buffer address to which data will be transferred (on a read), or from which data will be transferred (on a write). Users specify this address in the system call packet. When the system requests that a system buffer be read, the buffer header address assigned by Buffer Management, is stored here.

The disk controller transfers data directly to the location specified by the caller. If the data transfer request is initiated by the system, the data buffer address will be in ring 0, always resident and mapped logical to physical. If the request is issued by the user, there is the strong possibility that the user buffer will not be resident when the transfer is taking place. Therefore, user buffer must be faulted and pinned before the disk request is enqueued, and unpinned after the request completes. The number of pages spanned by the buffer is stored in the CCB at offset CBNPG.

Finally, the type of disk request must be specified in the CCB. The CCB command (request type) is stored at offset CBFLG in bits 13-15. The valid CCB commands were described previously in Section 3.3.1. Bits 0-7 correspond to the rings in the caller's address space in which a shared read has been done (?SPAGE). Bits 8 and 9 are defined as follows:

CBPIO (8) = Physical I/O Bit (set at open)
CBFWE (9) = Write access to file bit (set at open)

Bits 10. through 12. are undefined.

Three CCB fields are used by CCB Request Management during IOCB processing to facilitate the I/O procedure by eliminating unnecessary index block retrieval and examination. Although index blocks of all files are enqueued to the FCB buffer list when they are read in from disk, traversal of this queue on subsequent requests may sometimes be bypassed by saving the logical disk addresses and the offsets into index blocks in the CCB of the last requestor.

Consider the following situation in which data block 6 is requested from file FOO, whose elementsize is equal to 4. As CCB Request Management processes the request, the logical disk address of the data element, as well as the offset into the first-level index block, are saved in the CCB. Suppose that the caller then makes a subsequent request for data block 7. Since this request will access the same data element as the previous request (offset into first-level index block matches), the index block need not be read. Furthermore, the logical disk address of the data element is already available in the CCB and can be read immediately, without CCB Request Management touching an index block. Two- and three-level indexed files use CCB parameters to a greater extent to further expedite the data transfer by "skipping over" the same index blocks read in on the previous data transfer. However, the lowest-level index block will always be read in or found on the FCB buffer list queue in files with index levels greater than 1. This entire process is called INDEX LEVEL OPTIMIZATION. The pertinent CCB parameters are used as follows, relevant to the last disk I/O request on the file.

CCB Parameter	1 index level	2 index levels	3 index levels
CBIAH.W	LDA data element	LDA level 1 index block	LDA level 1 index block
CBXIA.W	Not used	Not used	LDA level 2 index block
CBIBN	Offset into level 1 index block	Offset into level 2 index block	Offset into level 3/2 index block

Word offset CBIBN, which contains the index block offset accessed on the last request, is actually divided into two fields. Bits 8-15 contain the logical offset (range 0-127) of either the level 1 or level 2 index block. If the file is a three-level file, bits 0-7 contain the logical offset of the level 3 index block. Otherwise, bits 0-7 are zero. This entire "index level matching" procedure will be explained further in CCB Request Management.

The file's parent directory CCB pointer and its FIB intra-directory pointer must be stored in the CCB as well. They are initialized at file open time at CCB offsets CBPCB.W and CBFIB, respectively. The parent CCB is needed for any I/O on the file's various associated directory data elements (e.g., FAC for access control list).

The CCB status word, CBSTS, is used for several purposes. If the file is a unit type file, bits 11-15 contain the unit type. The unit type is accessed by Unit Management as an index into pre- and post-processing dispatch tables. Unit CCBs maintain individual unit I/O request data, so the CCB is the appropriate database in which to store the unit type. The CCB definitions for unit types are the following:

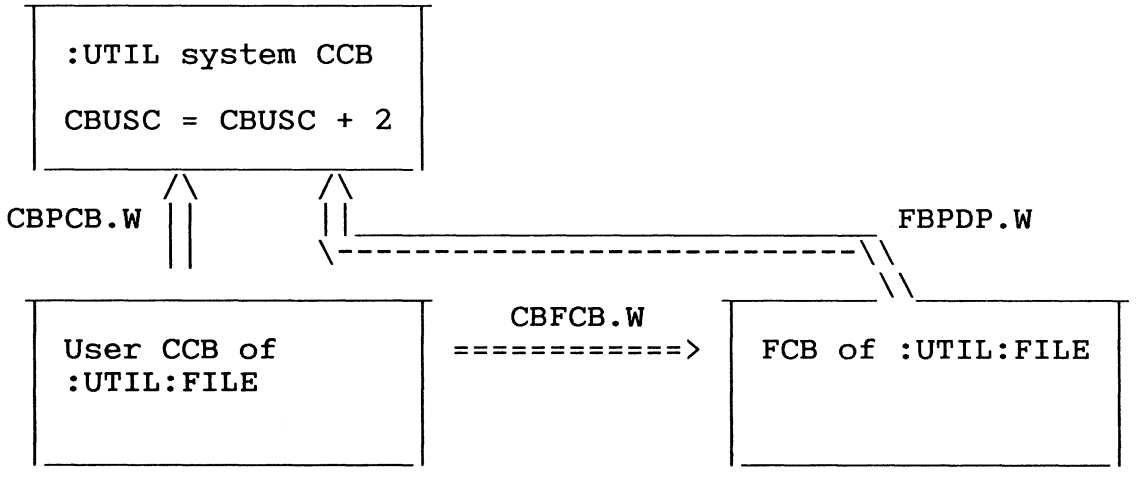
```
CUTMT (0) = Magnetic Tape Unit
CUTMC (1) = MCA Unit
CULPB (2) = Line Printer Unit
CULPD (3) = LP2 Printer
CULPE (4) = Laser Printer
```

Disk units opened separately as unit type files (not LDUs) are treated as a single unit LDU with no invisible space. "Physical disk I/O" is done instead of "unit I/O." For this reason, there is no disk unit type defined in the CCB. If the file is a disk file, bits 11-15 contain the access privileges allowed the caller, which are extracted from the file's ACL (FAC). Bits 0-10. of CBSTS have the following definitions:

CCB Status Bit	Function
CBUNT (0)	Unit type CCB (no disk units). Set on open of unit type files.
CBERB (1)	Error occurred. Set during IOCB processing if error occurs. Error code saved in CCB offset CBERR.
CBFSH (4)	File is shared. Set on shared open of file.
CHSHB (5)	I/O request is shared. Set prior to enqueuing shared I/O request.
CBRNA (6)	Read Next Allocated block request (disks only) Set prior to enqueuing RNA request (?BLKIO).
CBSAF (7)	?WRB with ?SAFM option request (Mag Tapes). Set prior to enqueuing unit I/O request.
CBMIO (8)	Modified sector I/O (disks only). Set prior to enqueuing mod sector I/O (?BLKIO).
CBEOV (8)	Enable VFU load (Line Printers). Set prior to enqueuing I/O to LPU types.
CBPEB (9)	Peripheral type CCB. Set on open of generic/IPC type files.
CBIFF (10)	Inhibit initial form feed (Line Printers). Set prior to enqueuing I/O to LPU types.
CBFNS (10)	File number set (Mag tapes). Set prior to enqueuing I/O to mag tape and MCA.

The CCB unique ID (offset CBUID) is set with the same value as the file's FCB unique ID (offset FBUID). When a disk I/O request is made, the CBUID is compared to FBUID of the same file. If the unique IDs do not match, AOS/VS will panic with code 6034. The IDs should always match! Inconsistent unique IDs usually mean that either the CCB or FCB memory has been corrupted, the causes of which can possibly be discovered through analysis of the system's memory dump.

There is one circumstance in which multiple file openers use the same CCB for a file: when the file opener is the system opening a directory type file in Resolution Services. If 20 processes open :UTIL:FILE, each process maintains a unique CCB for FILE in ring 1 of its address space, but the system keeps only one copy of the system CCB for UTIL. A count of the number of objects using the system CCB is maintained in a redefined CCB parameter for directory files. This parameter is the CCB use count, CBUSC. Since the number of blocks specified for any directory file I/O request is 1, CBUSC replaces CBNBK. The CCB use count is incremented when it becomes the parent of a new FCB, specifically, when FBPDP.W of a subordinate file in the directory is set with the CCB address. The CCB use count is incremented again when it becomes the parent of the CCB of the same file, specifically, when CBPCB of the subordinate file is set. This implies that the use count indicates the number of objects, not processes, that literally point to the system CCB. See the following illustration.



Finally, like numerous other AOS/VS static databases, the CCB is locked during certain critical region operations. The CCB lock word in which four lock bits are defined is CBLOCK. The lock bits are:

Bit position in CBLOCK	Bit position in CCB	Function
CBTRAN (0)	BFBTRAN	CCB transition lock
CBLKB (1)	BCBLK	CCB vanilla lock
CBTPB (2)	BCBPL	CCB lock waiter bit
CBFLB (3)	BCBFL	CCB fault lock

File Management provides four CCB locking services that uniformly acquire locks properly and call the execute pending mechanism if the lock is already held. The standard system CCB locking services are the following:

CCB Lock Service	Function
CLOCK/ CULCK	Lock/Unlock a system CCB. Path will pend if lock already held.
UCLOCK/ UCULCK	Lock/Unlock a user CCB. Returns "Simultaneous requests on same channel" if lock already held.
PGCBLK/ PGCULK	Lock/Unlock a page file (system) CCB. Path will pend if lock already held.
FCLOCK/ FCULCK	Lock/Unlock a CCB when during a page fault on the file. Path will pend if lock already held.

Some code paths do not use these services, but attempt to set a lock themselves. Such situations are acceptable when a specific, immediate action must be taken if the lock is not available. For example, RDB.P returns "simultaneous request on same channel" if the vanilla lock is set (user CCB); RESLV.P simply sets the bit when initializing a newly created system CCB.

The main purpose of the CCB transition lock is to gain exclusive access to the CCB for a very short time. The transition lock is a spin lock, which is always acquired before setting either the vanilla (long-term) CCB lock or the CCB fault lock. This imperative action is taken by all of the CCB locking services. Once the desired lock is held, the transition lock will be released. It is important to note that a code path, which has acquired the transition lock, never pends. Another code path attempting to gain access to the same lock will spin and potentially hang the system by hogging the CPU. The CCB transition lock will be released before pending.

The CCB vanilla (or pend) lock is acquired when the calling code path must perform extensive operations on the CCB and/or if the caller runs the risk of pending. For example, system CCBs of directory type files must be locked especially when the use count (CBUSC) is modified. When Resolution Services resolve a pathname, the system CCB of each directory is locked. The use count is incremented to indicate to Close Services that the file is still in use (FCB cannot be destroyed, file cannot be deleted); the CCB is later unlocked when the next file in the pathname (its son) is opened.

All CCBs are locked before the CCB request is enqueued for I/O and remain locked during the I/O processing. In most cases, the caller is responsible for unlocking the CCB after the I/O completes. The exceptions are user logical disk read and write requests. Since no further system processing is necessary when these requests complete (caller's control block has even been released already), the CCB Post Processor conveniently unlocks the CCB.

The CCB fault lock is a special lock used by Memory Management when a page must be faulted in from the file. FCLOCK must first acquire the CCB vanilla lock before setting the fault lock and then hold both locks until the page fault is over. This is to prevent code paths that do NOT call CLOCK or UCLOCK, but set BCBLK explicitly, from accessing a CCB with a page fault in progress. In fact, when CLOCK acquires the vanilla lock, it sanity checks to verify that the fault lock is not set. If it is, a panic 6355 will occur.

If any of the CCB locking services attempt to acquire a lock that is currently held, the control block will be pended via a call to the Process Management service MPEND. Before this occurs, the CCB pend bit must be set. This will indicate to the CCB unlocking services that at least one waiter exists for a CCB lock. The unlocking service will clear BCBPL and call UNPEND with the CCB address as a pend key upon releasing the lock, which will place all pended waiters for the CCB on ELQUE. The next control block to be scheduled will then successfully acquire the desired lock.

All of the CCB locking/unlocking services are concise. Observe how CLOCK is implemented:

```
CLOCK (CCB_address);            /* Input - addr of CCB to lock */
{
TOP:
  /*************************************************************
   * Get CCB transition lock.                                  *
   * This is done by calling the base level transition (spin) *
   * lock service BSLOCK.  Spin until lock is free.            *
   *************************************************************/

     call BSLOCK (CCB_addr, BCBTRAN);

  /*************************************************************
   * If ESD is running, the system is being shut down.         *
   * Many files are probably open.  ESD calls CLOCK as         *
   * standard practice for acquiring the vanilla lock to       *
   * manipulate the CCB.  If it is already locked, clear all   *
   * other locks and "give" ESD the lock.                      *
   *************************************************************/

     if (ESD running)
        {
        clear_bit (CCB_addr, BCBTRAN);   /* Clear transition  */
        set_bit   (CCB_addr, BCBLK);     /* Give ESD vanilla  */
        clear_bit (CCB_addr, BCBPL);     /* I say no waiters  */
        clear_bit (CCB_addr, BCBFL);     /* I say no faulters */
        return;                          /* ESD will be happy */
        }

     /*********************************************************
      * Use atomic instruction to attempt locking the CCB.    *
      *********************************************************/

     if ( check_bit (CCB_addr, BCBLK) )
        {

     /*********************************************************
      * Vanilla lock is already set.                          *
      * Set lock waiter bit and pend until woken up by some   *
      * CCB unlock service.  Then try for vanilla lock again. *
      *********************************************************/
```

```
         set_bit   (CCB_addr, BCBPL);
         call MPEND (BCBTRAN, CCB_addr);
         goto TOP;
         }

      else
         {
      /*********************************************************
       * Good.  The WSZBO set the vanilla lock bit!            *
       * Now check the fault bit, which should not be set.     *
       * If it is, panic.                                      *
       *********************************************************/

         clear_bit (CCB_addr, BCBTRAN);  /* Release trans lock */

         if (check_bit (CCB_addr, BCBFL)
            call PANIC (6355);

         return;                         /* CCB locked: return */
         }
}
```

3.3.3 CCB Creation/Destruction

System and internal CCBs are allocated directly from general system memory (GSMEM) either "implicitly" by Resolution Services or "explictly" by File Open Services.

When Resolution Services creates a CCB, the file is always a directory type file, and the CCB is used exclusively by Resolution Services when resolving pathnames. More importantly, it is used to keep track of the number of openers of subordinate files in the directory. Furthermore, the system CCB pointer in the FCB WILL contain the CCB address of this implicitly opened file, and the file open count in the FCB will NOT be incremented.

When Open Services creates an internal CCB, explicitly for the system via an internal open call (IOPEN.P, XEOPEN.P, etc.), the procedure implies that the system is actually acting as a user. However, since the AOS/VS kernel exists only in ring 0, so must the CCB. The system CCB pointer in the FCB will NOT contain the CCB address of this explicitly opened file, and the file open count in the FCB WILL be incremented. The latter procedure is followed for user CCBs created in Open Services as well. Files opened explicitly by the system are swapfiles, pagefiles, breakfiles, program files, and IPC spool files.

Although user CCBs are handled in the same way, allocation of user CCBs is not as straightforward as direct memory allocation from GSMEM. Since a process may be swapped out during memory contention, user CCB pages must be allocated and mapped to the swappable portion of the process address space. AOS/VS chooses to map these pages contiguously to a fixed ring 1 address beginning at the global entry point CCBTAB, whose value is 02000100000. File Management always attempts to allocate CCBs on resident pages first.

The maximum number of CCBs that fit on a page is:

PAGESIZE/CCBLT = 1024./32. = 32. CCB/page.

The maximum number of channels available to a user is 256., one of which the local PMGR always uses. A CHANNEL is simply a number that provides File Management with a means of quick and easy access to the CCB associated with a given file opener. The operation that converts a channel number into a user CCB address, GENCCBAD, will be discussed later. The total number of user CCB pages possible is then 8. The process table extender keeps track of the total number of user CCBs in use (open files) for a process. Eight words beginning at process table extender offset PUCCBS contain the number of CCBs in each user CCB page:

```
PEXTN->PUCCBS[0] = number of CCBs in use on user CCB page 0
PEXTN->PUCCBS[1] = number of CCBs in use on user CCB page 1
PEXTN->PUCCBS[2] = number of CCBs in use on user CCB page 2
PEXTN->PUCCBS[3] = number of CCBs in use on user CCB page 3
PEXTN->PUCCBS[4] = number of CCBs in use on user CCB page 4
PEXTN->PUCCBS[5] = number of CCBs in use on user CCB page 5
PEXTN->PUCCBS[6] = number of CCBs in use on user CCB page 6
PEXTN->PUCCBS[7] = number of CCBs in use on user CCB page 7
```

A channel number corresponding to the relative location of the opened file's CCB is assigned and returned to the calling process for subsequent reference to the file. For instance, the first user CCB is created at CCBTAB and assigned channel number 0. The next CCB is created at CCBTAB + CCBLT and assigned channel number 1. PEXTN->PUCCBS[0] will contain 2, the number of CCBs in use on CCB page 0. (The first file opened by any user process will never be assigned a channel number of 0 because the local PMGR assigns channel 33. to open PMGR.SF, causing CCB page 1 to become resident. OFAULT.P will consequently assign the next 31. CCBs from page 1 before assigning CCBs from CCB page 0. See Section 3.3.4.)

Now is a good time to explain the terminology of CCB creation and destruction. User CCBs are not allocated; their pages are allocated. System and internal CCBs are not allocated; their memory is allocated. Therefore, databases such as CCBs and FCBs are actually "created" and "destroyed" within blocks of system memory. The terminology of "allocating" and "releasing" databases is used quite loosely throughout AOS/VS code and comments. This manual attempts to clarify this distinction.

3.3.4 CCB Operations: OFAULT.P

The File Management operation that allocates user CCB pages, creates the CCBs, and assigns channel numbers is called OFAULT.P. OFAULT.P is called only from File Open Services, namely SOPPF.P and GOPEN.P, which must create user CCBs. The caller may wish OFAULT.P to assign a channel number dynamically, or to assign a specific channel number. The new CCB address will be returned. Following is the C-based algorithm that illustrates the AOS/VS implementation of OFAULT.P.

```
#define CCBPERPG 32                          /* = 1024./CCBLT     */
#define SCNCCBPG 7                           /* Num CCB pages     */
/* Input: specific chan num to assign or -1 for system assign */
/* Output: assigned channel number, CCB address                */

OFAULT.P (chin, *chout, *CCB_addr)
{
/*************************************************************
 * Create a user CCB.  Determine type of request.            *
 *************************************************************/

   if (chin != -1)       /* Static channel request !!! */
      {
   /*********************************************************
    * Given the channel number, generate the user CCB address.*
    * Fault and pin the CCB so page remains resident.        *
    *********************************************************/

      call GENCCBAD (chin, CCB_addr);
      call FLTPIN (CCB_addr);

   /*********************************************************
    * Test CCB validity.                                    *
    * If CCB locked, another request is in progress. Error  *
    * out! The WSZBO sets the CCB lock bit if not set before. *
    *********************************************************/

      if (WSZBO (CCB_addr, BCBLK))                /* Sim reqs? */
         {                                           /* Yup */
         call UNPIN (CCB_addr);                    /* Adios */
         return (simultaneous_requests_on_same_channel);
         }

   /*********************************************************
    * If the process table address is non-zero, the channel is*
    * already in use.  Error out!                           *
    *********************************************************/

      if (CCB_addr->CBPTA.W != 0)          /* Channel in use?  */
         {
         clear_bit (CCB_addr, BCBLK);                /* Yup */
         call UNPIN (CCB_addr);                      /* Bye */
         return (channel_in_use);
         }
```

```
/*************************************************************
 * CCB valid.  Calculate CCB page number and bump the count*
 * of CCBs in use for this CCB page in the PTBL extender.  *
 * Then ready it for caller by zeroing it out and initting *
 * it locked.                                              *
 *************************************************************/

   CCB_page = chin/CCBPERPG;          /* Compute CCB page num */
   PEXTN->PUCCBS [CCB_page] += 1;     /* Bump num CCBs on page*/

   zero (CCB_addr, CCBLT);            /* Clear out the CCB    */
   set_bit (CCB_addr, BCBLK);         /* Init it locked       */
   return;                            /* DONE for static req  */
   }

else                    /* dynamic channel request !!! */
   {
/*************************************************************
 * Each CCB page is checked for free CCB.                  *
 * Pass1 = resident CCB pages are checked for free CCBs.   *
 * Pass2 = no resident CCB pages; new pages are allocated. *
 *************************************************************/

   for (pass = 1;  pass <= 2;  pass++)
      for (CCB_page = 0;  CCB_page < SCNCCBPG; CCB_page++)
         {
         /* Any free CCBs on this page?                     */
         if (PEXTN->PUCCBS [CCB_page] < CCBPERPG);
            {
         /*************************************************
          * CCB logical page addr begins at CCBTAB.  Shift *
          * the CCB page to the left of the offset field.  *
          * Adding them gives log start of CCB page.       *
          * (Residency check done via LPHY instruction)    *
          *************************************************/

            CCB_page_la = CCBTAB + (CCB_page << 10.);
            if (resident (CCB_page_la) || (pass == 2))
               {
         /*************************************************
          * If pass1, page is resident.  FLTPIN pins it.   *
          * If pass2, page may or may not be resident.     *
          * Fault and pin it.                              *
          *************************************************/

               call FLTPIN (CCB_page_la);
```

```
        /*************************************************
         * Check each CCB in page (ccbip) to make sure it *
         * is not in use.  If it is not, clear it, set the*
         * vanilla lock (to indicate channel in use),     *
         * bump the CCB in use count for the page in      *
         * PEXTN, and calculate and return the channel    *
         * number.                                        *
         * The caller will unpin the CCB page!            *
         * If no free CCBs are found in the page, unpin   *
         * it and search next page!                       *
         *************************************************/

              for (ccbip = 0; ccbip < CCBPERPG; ccbip++)
                 {
                 CCB_addr = CCB_page_la + (ccbip * CCBLT);

                 if (!bit_already_set (CCB_addr, BCBLK) &&
                     (CCB_addr->CBPTA.W == 0))
                   {
                   /* Channel Not In Use!  Grab The CCB!  */
                   set_bit (CCB_addr, BCBLK);
                   CCBS_in_use = PEXTN->PUCCBS [CCB_page]++;
                   *chout = (CCB_page * CCBPERPG) +
                                                CCBS_in_use;
                   return;
                   }
                 } /*end of for */

        /*************************************************
         * No free CCBs on this page.                     *
         * Unpin the page and search the next page.       *
         *************************************************/

                 /* No free CCBs on this page. Release it */
                 call UNPIN (CCB_page_la);

              } /* end of if resident */
            } /* end of if above that */
          } /* end of inner for */
        } /* end of outermost for */

     /*********************************************************
      * Pass 1 and 2 done and no free CCBs found.             *
      * This means the hog has exhausted all 255. channels!   *
      *********************************************************/

       return (no_free_channels);  /* Tell user he is a hog */

    } /* end of else */
  } /* end of OFAULT.P */
```

3.3.5 CCB Operations: Generate CCB address (GENCCBAD)

The File Management operation that calculates a ring 1 user CCB address from a channel number is called GENCCBAD. GENCCBAD is called by OFAULT.P when a specific channel is requested on a file open. It is called by the DFAULT operation to calculate the CCB address of an input channel number on behalf of a system call servicing a user request. GENCCBAD is called by Memory Management to retrieve the CCB address of a shared file whose channel number is found in a shared page's control directory entry (CDE).

Since channel numbers are assigned in the same order that user CCBs are created, the address calculation is a simple operation. The base address in ring 1 where user CCBs are found is CCBTAB. The CCB page number multiplied by the size of a page, 02000 words (shifted left 10.), added to CCBTAB yields the logical address start of the CCB page. The number of the CCB in the page multiplied by the CCB length gives the offset into the page of the desired CCB.

The sole input to the GENCCBAD is the channel number and the sole output is its logical ring 1 address. GENCCBAD assumes that the process whose CCB address is being found is currently mapped. The following algorithm illustrates GENCCBAD.

```
GENCCBAD (channel_num, *CCB_addr)
{
/*************************************************************
 * Get the CCB page and the number of the CCB in the page.    *
 * Concretely, if the channel number is 48, 48/32 = 1 rem 16. *
 * The CCB page is 1, the CCB is the 16th one in page.        *
 *************************************************************/

  CCB_page = channel_num / CCBPERPG;
  CCB_in_page = channel_num % CCBPERPG;

/*************************************************************
 * Calculate the CCB address and return it.                   *
 * If the channel number if 48, the CCB address is:           *
 *           02000100000 + 02000 + 01000 = 02000103000        *
 *************************************************************/

  *CCB_addr= CCBTAB + (CCB_page << 10.) + (CCB_in_page * CCBLT);
  return;
}
```

3.3.6 CCB Operations: DFAULT and RUCCB.P.

The File Management operation that returns the ring 1 user CCB address given a channel number is called DFAULT. This operation is called from every File Management system call operation whose input from the user is an open channel number. The channel number is translated into the corresponding user CCB address via a call to GENCCBAD, and the CCB process table offset is verified for a non-zero value, indicating the user indeed has the channel open. Some system calls retrieve the CCB address from DFAULT and prepare and enqueue the returned CCB for user I/O, e.g., ?RDB, ?GTRUNC. Other system calls use the returned CCB address only to access the parent directory CCB and initiate I/O on the parent directory, e.g., ?RENAME, ?CPMAX.

The File Management operation that destroys a user CCB is called RUCCB.P, Release User CCB. RUCCB.P simply takes the input channel number, generates the user CCB address via a call to GENCCBAD, and marks the CCB as free (not in use) by clearing the vanilla lock bit (BCBLK) and zeroing the process table offset (CBPTA.W). In addition, the CCBs in use count for the CCB page, beginning at offset PUCCBS in the process table extender, must be decremented. After the user CCB is released, the caller will request KFCB.P service. This initiates the chain reaction that destroys the chain of system CCBs/FCBs, which must remain open from the root directory down to the file opened on the user CCB. RUCCB.P is called by GCLOSE.P, and by open services on errors occurring after the user CCB's creation.

3.3.7 CCB Operations: Kill Channel Control Block (KCCB.P)

KCCB.P is the operation called to destroy system CCBs when a file is being closed. KCCB.P is called from those modules that create system CCBs and must close them. KCCB.P destroys the CCB, then checks the associated FCB open count. If it is zero, the FCB will be destroyed. Moreover, the parent CCB of this hierarchical level is checked for a zero use count. If there are no more users, it will be released as well. KCCB.P will loop, working its way through the directory hierarchy, destroying all system CCBs and FCBs (if CBUSC is 0) until it gets to the root. Of course, the root CCB can only be destroyed at system shutdown.

This operation does not "call" KFCB.P to destroy FCBs, but executes overlapping assembly code common to both KCCB.P and KFCB.P. Consequently, some algorithmic code is duplicated from KFCB.P in the following illustration of KCCB.P. The only input to KCCB.P is the system CCB address (which must be locked upon entry).

```
KCCB.P (CCB_addr);
{
/*************************************************************
 * The use count has already been decremented.  If it is still*
 * in use, it cannot be destroyed.  Just unlock it and return.*
 *************************************************************/

   if (CCB_addr->CBUSC != 0)
      {
      call CULCK (CCB_addr);
      return;
      }

/*************************************************************
 * If there are no more users of the CCB, destroy it and     *
 * release its memory.  Procedure: lock the parent, decrement *
 * the parent CCB use count (one less son), unpend any waiters*
 * on the CCB lock, and finally deallocate it.  There can be  *
 * only one situation in which a control block can be pended  *
 * awaiting a lock when the use count is 0: resolution        *
 * services.  See the note following the algorithm!           *
 *************************************************************/
```

```
KCCB_LOOP:

   PCCB_addr = CCB_addr->CBPCB.W;       /* Save parent CCB addr*/
   FCB_addr = CCB_addr->CBFCB.W;        /* Save FCB addr       */

   call CLOCK (PCCB_addr);              /* Lock parent CCB to  */
   PCCB_addr->CBUSC -= 1;               /* decr use count (CCB */
                                        /* to be destroyed!)   */
   if check_bit (CCB_addr, BCBPL)
      call UNPEND (CCB_addr);           /* Unpend any waiters  */
                                        /* before ...          */
   call RSMEM (CCB_addr, CBBLT);        /* CCB IS RELEASED!    */

   FCB_addr->FBSCB.W = 0;               /* No more system CCB  */
   if (FCB_addr->FBOPN != 0)            /* Check FCB open count*/
      {                                 /* File is still open  */
      call CULCK (PCCB_addr);           /* Unlock parent CCB   */
      return;                           /* Rest of hierarchy   */
      }                                 /* stays open for now. */

/*************************************************************
 * Last opener is closing file.  Clean up the FCB.           *
 * Release buffers on FCB buffer list.                       *
 * Release shared pages on FCB.                              *
 * KFCB.P ALSO FOLLOWS THIS CODE PATH!                       *
 *************************************************************/

   call RELBF (FCB_addr);               /* Release FCB buffers */
   if (FCB_addr->FBCMP.W != -1)         /* and shared pages if */
      call RELSP.P (FCB_addr);          /* there were any.     */

/*************************************************************
 * If the file was marked for delete on last close, delete it *
 * now!  Then jump to KCCB to release the system CCB/FCB      *
 * hierarchy above.                                           *
 *************************************************************/

   if check_bit (FCB_addr,BFBDL)
      {
      call CLDEL.P (FCB_addr->FBFIB, PCCB_addr, FCB_addr);
      goto KCCB_LOOP;
      }

   /* If unit type file, do unit-specific operations */
   if (FCB_addr->FBTYP == UNIT_TYPE_FILE)
      call UCLOSE.P;
```

```
/*************************************************************
 * Read in the FIB, update it with the modified data from the *
 * FCB's Funny FIB and flush it out to disk NOW.  Flushed now *
 * (RELF) because no other users have file open.              *
 *************************************************************/

   call RAID (FCB_addr->FBFIB, DEFIB, PCCB_addr,
              &FIB_BH, &FIB_addr);

   FIB_addr->FIFCB = 0;
   call UPFIB (FIB_addr, FCB_addr);
   call RELF (FIB_BH);

/*************************************************************
 * Almost done.                                              *
 * Release CPB memory.  Destroy the FCB!                     *
 *************************************************************/

   call RSMEM (FCB_addr->FBCPB.W, CPBLT);  /* Release CPB mem */
   call RFCB (FCB_addr);                    /* Annihilate FCB  */

/*************************************************************
 * Now decrement the parent dir CCB use count.               *
 * If still users, just unlock it and return.                *
 * If not more users, it can be released (by KCCB.P), which  *
 * will execute this entire routine again, and possibly      *
 * destroy all open CCBs/FCBs open in the pathname, up to the *
 * root.                                                     *
 *************************************************************/

   if (--PCCB_addr->CBUSC)                 /* One less FCB son */
      {                                    /* PCCB use count 0?*/
      call CULCK (PCCB_addr);              /* No, unlock it,   */
      return;                              /* leave hierarchy. */
      }

   CCB_addr = PCCB_addr;                   /* Yes, make parent */
   goto KCCB_LOOP;                         /* current CCB and  */
                                           /* destroy it.      */
} /* end of KCCB.P */
```

There is one condition in which the system CCB use count (CBUSC) may be 0 when another CB is pended awaiting the release of the CCB lock. KCCB.P is called to release a system CCB once the caller has finished using it. Callers lock the CCB, decrement the use count (if they had incremented it) and call KCCB.P. KCCB.P checks the use count. If it is non-zero, the CCB is just unlocked. If it is 0, the CCB memory will be deallocated and its pointer in the FCB (still there) will be cleared. However, before making the call to RSMEM, KCCB.P calls UNPEND to unpend waiters of the CCB lock. It would seem possible, then, for the waiter to be awoken, the CCB will become deallocated and its address become invalid, and the waiter will have an invalid address which may panic the system. AOS/VS is careful to avoid this condition.

The waiters of system CCB locks, when the use count is 0, can only be control blocks whose code paths are in Resolution Services, which provide a special mechanism to protect against the previously described potential panic situation. Any file system request whose input is a pathname calls Resolution Services to retrieve the system CCB. If the system CCB is locked, Resolution Services waits for the lock.

Resolution Services protects against using a potentially bad CCB address upon being unpended by checking the system CCB pointer in the FCB. KCCB.P zeroes out FBSCP.W, so Resolution Services will know that the system CCB it was awaiting has disappeared. Usually when a CB is unpended when awaiting a lock, another immediate attempt at acquiring the lock is made. However, Resolution Services must FIRST check that the CCB is still valid, because if it doesn't and the CCB was deallocated, the subsequent attempt at checking the lock bit will probably result in ring 0 memory corruption.

3.4 File Management Services

3.4.1 (System) Read in a Block: BLKIN.

BLKIN and its variants, CBLKIN and BLKINW, are the File Management services that read in one specific file block into a system buffer. BLKIN also takes the liberty of searching the FCB buffer list for the buffer header containing the data. This may eliminate the need for actual disk I/O. If the buffer header is not found, BLKIN will initiate the disk I/O and return the buffer header address to the caller.

The caller of BLKIN may or may not modify the data contained in the system buffer returned. Either way, the buffer must be released to the free buffer chain BFLRU.W when the caller is done with it (if there are no more users and no waiters). If the caller modifies the data, the buffer must either be released modified or flushed immediately. If the caller does not modify the data, the buffer will either be released normally or destroyed. See Buffer Management for a description of these actions.

The following are the variants of BLKIN:

Variant	Function
BLKIN	Read one block from a file into a system buffer. Called by File Mgmt routines.
CBLKIN	Read one block from a file into a system buffer. Called by non File Mgmt routines.
BLKINW	Search the FCB buffer list for the desired block. Called by CCB Request Management.

File Management routines call BLKIN and need only supply the CCB address of the file on which the I/O is to be performed. The caller must store the desired block number at CCB offset CBDBH.W. File Management routines use BLKIN to read directory data blocks and the directory bit map for all directory file I/O. The buffer headers for these blocks are enqueued to the FCB buffer list until released.

External system components call CBLKIN and must supply both the block number to read and the CCB address. Because File Management assumes that external components need not be concerned with CCB management, CBLKIN permits the caller to pass the block number, which it kindly stores in the CCB. The buffer headers for these "system" data blocks are enqueued to the FCB buffer list until released.

CCB Request Management calls BLKINW simply to check if a specific block is found on the FCB buffer list. If it is not, BLKINW will not read in the block, since CCB Request Management routines cannot call base level pending routines. An error

return from BLKINW will indicate that the block was not found in memory. CCB Request Management will have to initiate the disk I/O itself in order to correctly pend the IOCB. The callers of BLKIN and CBLKIN are listed below:

Callers of BLKIN

Caller	Effect
JELLO.P	Reads the directory bit map when allocating directory data blocks.
XINIT.P	Reads in the DIB when initializing an LDU.
DRLSE.P	Reads in the DIB when releasing an LDU.
DELETE.P	Reads in FIBs of all files subordinate to the directory being deleted. All subordinate files must be deleted as well.
LOOKUP.P	Reads each FNB block to search for filename (called by RESLV.P).

Callers of CBLKIN

Caller	Effect
IS.REC	IPC spoolfile I/O.
ISEN2.P	IPC spoolfile I/O.
CLNUP.P	IPC spoolfile I/O.
CHIFE.P	HIF file I/O (create HIF entry).
DHIFE.P	HIF file I/O (delete HIF entry).
HNAME.P	HIF file I/O (get host name/id).
SPNAM.P	PIF file I/O (set process name).

The inputs and outputs of BLKIN/CBLKIN are the following:

Variable	Input	Output
AC0	Not used.	Unchanged.
AC1	Block number (CBLKIN only).	Unchanged.
AC2	CCB address.	Buffer header address.

Finally, the C-based algorithm for CBLKIN is illustrated below. Notice that CBLKIN calls an even lower File Management service, NQCRQ, to enqueue the CCB. Since the callers of BLKIN/CBLKIN can be either external components or internal File Management services, NQCCB is not called directly.

```
CBLKIN (rel_blk_num, CCB_addr, *BH_addr_out)
{
/*************************************************************
 * Ensure that the CCB is valid.                             *
 * VALCID checks FBUID/CBUID match and panics with code 6032 *
 * if they are not the same.  Afterwards, get input relative *
 * file block number and store in CCB.                       *
 *************************************************************/

   call VALCID (CCB_addr);
   CCB_addr->CBDBL = rel_blk_num;

/*************************************************************
 * Get the FCB address from the CCB and lock the FCB.        *
 * FCB locked to prevent other JPs from FCB access in search! *
 * Then search FCB buffer list for the buffer header.  The   *
 * search key is the relative block number.                  *
 *************************************************************/

   FCB_addr = CCB_addr->CBFCB.W;
   call FXLOCK (FCB_addr, BFBTRAN);

   BH_addr = 0;
   search (FCB_addr->FBBLP.W, BQDBN, rel_blk_num, &BH_addr);

   if (BH_addr != 0) /* BH found! */
      {
    /*********************************************************
     * GREAT! BH found on FCB!                               *
     * Search is over, so unlock FCB and (transition) lock BH.*
     *********************************************************/

      clear_bit (FCB_addr, BFBTRAN);          /* Unlock FCB */
      call FXLOCK (BH_addr, BBQTRAN);         /* Lock BH    */

   /*********************************************************
    * If  no users/waiters on BH, it is on the LRU and      *
    * must be dequeued.  Otherwise, not on LRU.             *
    *********************************************************/

      if (BH_addr->BQRQC.W == 0)          /*  BH waiters/users? */
         call DQBCN (BH_addr);            /* No, deque from LRU */
                                          /* Yes, is not on LRU */
```

```
    /************************************************************
     * Flush BH if necessary.                                   *
     * If there is I/O currently in progress, BWAITS will       *
     * wait for the I/O to complete.  We must bump the          *
     * waiter count first so the BH post processor will know    *
     * to unpend us.  Upon return, we will have control of      *
     * the buffer, so bump the use count, unlock BH and         *
     * return the BH address to the user.                       *
     ************************************************************/

       BH_addr->BQWTC += 1;                  /* Bump wait count */
       call BWAITS (BH_addr);                /* while IOP on BH */
       BH_addr->BQWTC -= 1;                  /* Decr when done. */

       BH_addr->BQUSC += 1;                  /* We have the BH! */
       clear_bit (BH_addr, BBQTRAN);         /* Unlock xlock    */

       *BH_addr_out = BH_addr;               /* Return BH addr  */
       return;                               /* Return control  */
       }

    else  /* BH not found on FCB buffer list */
       {
     /************************************************************
      * Buffer header not on FCB.                                *
      * Must initiate I/O ourselves.  We must call NQCRQ since   *
      * we are a system-wide service, and NQCRQ handles system   *
      * I/O requests.  We build the packet and send it to        *
      * NQCRQ, which will do an NQCCB request and pend until     *
      * the I/O is complete.  The CCB command that BLKIN         *
      * always uses is read system buffer. (The BH addr is       *
      * returned in CBUAD.W of the CCB by post processor.)       *
      ************************************************************/

       clear_bit (FCB_addr, BFBTRAN);          /* Done with FCB */

       set_up_packet (nqcrq_packet);
       call NQCRQ (nqcrq_packet, CCB_addr);       /* Do the I/O */

       *BH_addr_out = CCB_addr->CBUAD.W.      /* Return BH addr */
       return;                                /* Return control */
       }

} /* end of CBLKIN */
```

3.4.2 Enqueue Channel Control Block Request (NQCCB)

NQCCB is the File Management operation that creates and initializes an I/O Control Block (IOCB), enqueues to the IOCB scheduling queue IORUN.W, and wakes up the disk manager control block to run a logical disk I/O request. IOCBs are enqueued to IORUN.W; CCBs are not enqueued anywhere under normal circumstances. If the system is operating under heavy memory contention conditions and a free IOCB cannot be allocated from the IOCB database pool, the CCB will actually be enqueued to the CCB wait queue CCBWQ.W, until an IOCB becomes free.

There are no variants of NQCCB. It is one standard operation custom-designed for all system call services that implement File Management system calls that do I/O on behalf of the user. The callers are the following system call services.

Callers of NQCCB

Caller	Effect
RDB.P WRB.P	Enqueues read request, CCB command CBRED. Enqueues write request, CCB command CBWRI.
SPAGE.P	Enqueues shared read request, CCB command CBRED, sets bit CBSHB in CCB status word.
DELETE.P	Enqueues delete file request, CCB command CBDEL.
GTRUNC.P	Enqueues truncate file request, CCB commands CBTRN1 and CBTRN2.
ALLO.P	Enqueues allocate file blocks request, CCB command CBALL.
NQCRQ	Enqueues CCB request on behalf of a "non" file system component.

NQCRQ, enqueue channel request, makes the NQCCB call for non File Management components which, for reasons of organizational design (modularity), are not permitted to modify the CCB request parameters to such an extent. NQCRQ takes two inputs from the system: a Channel ID (CID) and a packet address. The CID is the title given to the CCB of a file opened by the system as if the system were a user. For example, system initialization opens the :HIF, :PIF and :PER files and keeps them open forever. Swap files and page files are opened internally by the system as well. Although the system opens these files and allocates memory for CCBs, these CCBs are deemed channel identifiers. In short, the CID is the requesting CCB address.

The NQCRQ packet, set up by the caller, usually on the stack, contains exactly what the caller would have initialized the CCB with had he been privileged to store the parameters. NQCRQ moves the data from the packet into the CCB. Some special processing is done, however. For example, if the core manager task has called NQCRQ, the CCB is given the highest possible priority (0). Also, the CCB post-processor address is not a packet input, but NQCRQ determines the appropriate one based on the type of request.

NQCRQ commands have been defined as well. These commands are equivalent to their corresponding CCB commands, but a "CRQ" prefix replaces the "CB" prefix in the command parameter definition. After NQCRQ moves all the necessary information to the CCB, the request is enqueued to CCB Request Management via NQCCB. The following list of modules utilize the NQCRQ service.

NQCRQ Command	Caller	Effect
CRQRED	RDUST.P	Read UST from .PR file.
	JPMLOAD.P	Read in microcode file.
	RINGLD.P	Read in user page 0 of .PR file.
	READPR	Read unshared pages from .PR file.
	READPG	Read pages from user pagefile.
	SWAPIO	Read from user swapfile.
	IGHOST	Read page 0 of AGENT.PR.
CRQSHRD	READSH	Read shared pages (.PR/data file).
CRQSYB	BLKIN	Read one block from a file into a system buffer.
CRQWRI	WRBRK.P	Write to a breakfile.
	SWAPIO	Write to a user swapfile.
	UNFIO	Write to a user pagefile.

Since NQCRQ is just the system interface to NQCCB, only the latter will be demonstrated algorithmically. The inputs and outputs of NQCCB are the following:

Variable	Input	Output
AC0	CCB Post Processor address.	Unchanged.
AC1	Not used.	Unchanged.
AC2	CCB address.	Unchanges.

A typical calling sequence to NQCCB follows.

```
        XLEF    0,PPCBI          ;CCB post-processor (for CBSYB)
        XWLDA   2,CCBAD.W,3      ;Remember CCB addr from stack
        XJSR    NQCCB            ;Enqueue the request
        XJSR    CWAIT            ;Pend
```

CWAIT is the File Management operation, called after NQCCB requests, which pends the caller via a call to MPEND. The CCB post-processor will retrieve the requestor's CB from the CCB (offset CBTCB.W), explicitly dequeue it from PELEMQ and make it ready to run. If an error occurred during the request, the error code will be stored at CCB offset CBERR. CCB bit offset BCBER will be set as well. So, when the CB is unpended, either the error return or the good return will be taken, depending upon the state of BCBER.

Remember that the control blocks of ?RDB and ?WRB requests do not pend, but the task control blocks pend. The AGENT takes care of the TCB pending mechanism. However, File Management must "fool" the system call processor into thinking that the call has completed and the CB can be deallocated. This is done by zeroing the TCB address in the CB, making it appear to be a daemon instead of calling CWAIT. For example:

```
        XLEF    0,PPCUB          ;CCB PP (for ?RDB/?WRB)
        XWLDA   2,CCBAD.W,3      ;Remember CCB addr from stack
        XJSR    NQCCB            ;Enque the request
        XWLDA   2,CC.W           ;Caller's CB
        WSUB    0,0              ;Get ready to ...
        XWSTA   0,CATCB.W,2      ;... make CB a daemon
        XWADI   1,ORTN.W,3       ;Good return
        WRTN                     ;NOTHING ELSE TO DO!
```

Finally, the C-based algorithm, which describes this operation in detail, is illustrated below.

```
NQCCB (CCB_PP_addr, CCB_addr)
{
/*************************************************************
 * The CCB, if a user CCB, has a ring 1 address.  Since the   *
 * user's PTBL may not be mapped, we will convert the log     *
 * ring 1 addr to a physical addr.  Panic 6303 if error here! *
 *************************************************************/

   convert_to_physical_addr (CCB_addr);

   if (error)
      call PNIC (6303);

/*************************************************************
 * CCB OK.  Store passed CCB post processor address.          *
 * Clear the CCB error bit as well, as the call has not begun.*
 *************************************************************/
```

```
   CCB_addr->CBUPD.W = CCB_PP_addr;
   clear_bit (CCB_addr, BCBER);

/*****************************************************************
 * Allocate an IOCB.                                             *
 * If there is not enough memory, enqueue the CCB to CCBWQ.W     *
 * where it will wait until another CCB request finishes and     *
 * returns its IOCB to the pool.  CCBs are enqueued to           *
 * CCBWQ.W by order of priority.  Caller can gain control now    *
 * and pend.                                                     *
 *****************************************************************/

   IOCB_addr = 0;
   call DBALLC (&IOCB.DB, &IOCB_addr);

   if (IOCB_addr == 0)                          /* Enough memory?*/
      {                                         /* NOOOOOOOOOOO */
      call BSLOCK (IOWLK.W, BPDTRAN);           /* Lock CCBWQ.W  */
      enqueue (CCBWQ.W, CCB_addr, CBPRI);       /* CCB to CCBWQ  */
      clear_bit (&IOWLK.W, BPDTRAN);            /* Unlock CCBWQ  */
      return;                                   /* Caller pends  */
      }

/*****************************************************************
 * Do some accounting and initialize the IOCB for request!       *
 * IOAC = num currently active IOCBs.                            *
 * IOMX = max num active IOCBs since boot.                       *
 * Then enqueue the IOCB to IORUN by order of priority.          *
 *****************************************************************/

   if (++IOAC > IOMX)
      IOMX = IOAC;

   IOCB_addr->IOCCB.W = CCB_addr;               /* CCB phys addr */
   IOCB_addr->IOFCB.W = CCB_addr->CBFCB.W;      /* FCB addr      */
   IOCB_addr->IOSTW.W = IORDY;                  /* Status=ready  */
   IOCB_addr->IOSPC.W = &RUNRD;                 /* Init start PC */
   IOCB_addr->IONLV   = 0;                      /* Clear flg wrd */
   IOCB_addr->IOPRIO  = CCB_addr->CBPRI;        /* Priority      */

   call XLOCK (IOCBLK.W, BPDTRAN);              /* Lock IORUN.W  */
   enqueue (IORUN.W, IOCB_addr, IOPRIO);        /* Enque IOCB    */
   clear_bit (&IOCBLK.W, BPDTRAN);              /* Unlock IORUN  */

/*****************************************************************
 * IOCB is now enqueued to IORUN.W.                              *
 * Wake up the disk manager task, who will scan IORUN.W and      *
 * find, at least, this IOCB.  The request will be off and       *
 * running while the caller is pended.                           *
 *****************************************************************/

   call DWAKE;                                  /* BUZZZZZZZZZZ */
   return;                                      /* Caller calls  */
                                                /* CWAIT to pend.*/
} /* end of NQCCB */
```

3.4.3 File Open Services

File Open Services performs the function of opening a file. This service includes creating an FCB, if the file was not previously open, and always creating a CCB, whether the request is from the user or the system. The file's FIB must be read in from disk and copied into the FCB where the data is maintained while the file remains open. FCB and CCB parameters relevant to the calling process and the type of open being performed are initialized as well.

There are a number of variants provided in open services:

Variant	Function
GOPEN.P	Open a file for a user.
SOPEN.P	Open a file shared for a user.
ROPEN.P	Special AGENT open of a user file.
IOPEN.P	Internal system open of a file. The file pathname is in system space.
XIOPEN.P	Exclusive IOPEN.P.
EOPEN.P	Internal system open of a file. The file pathname is in user space.
XEOPEN.P	Exclusive EOPEN.P.

External requesting modules of GOPEN.P

Caller	Effect
User	?OPEN/?GOPEN - open a file. The AGENT converts normal ?OPENs to ?GOPENs.

External requesting modules of SOPEN.P

Caller	Effect
User	?OPEN/?SOPEN - Open a file. The AGENT converts ?OPENs with the ?SHOP bit set to ?SOPENs

External requesting modules of IOPEN.P

Caller	Effect
SINIT1	Open :PROC:PIF and :PROC:HIF files, open :SWAP and :PAGE directories, open IPC spool file for CLIBT, so it can receive an IPC from the PMGR.
IGHOST.P	Open AGENT.PR during ?PROC
PROC2.P	Open IPC spool file of PID being proc'ed.
BRKFL.P	Open breakfile.
LMCODE.P	Open microcode file to load daughter JPs.

External requesting modules of XIOPEN.P

Caller	Effect
OPENPU.P	Open each physical unit in the LDU to be initialized, if the name is in system space (:BOTH init).
ALLSW.P	Open PID's swap and page files on proc.

External requesting modules of EOPEN.P

Caller	Effect
PROPEN.P	Open program file for ?PROC, ?CHAIN, ?RINGLD.

External requesting modules of XEOPEN.P

Caller	Effect
OPENPU.P	Open each physical unit in the LDU to be initialized, if the name is in user space (standard LDU init).

The input values to each Open Service variant depend upon the variant called. The most common services requested are GOPEN.P and IOPEN.P. One input to GOPEN.P and ROPEN.P is the user packet, which is copied to the system stack. GOPEN.P must establish a trap handler in the active control block before user space is accessed to avoid a fatal error. The trap handler for GOPEN.P is GTRP, which will return a "System call parameter address error" if a protection violation occurs during the "WBLM" of user data to system space.

GOPEN.P is the only service that allows the caller to open IPC and generic files. The system has no reason to open these types of file, and any IOPEN.P that attempts to do so will receive an "Illegal file type" error.

Since IOPEN.P accomplishes essentially the same result as GOPEN.P without accessing user data, the C-based algorithm for the former will be described. IPC and generic file open code paths, which are actally simpler and shorter than other file type opens, will not be elaborated upon here. The inputs and outputs of IOPEN.P are the following:

Variable	Input	Output
AC0	Byte address of file pathname in sys space.	Unchanged.
AC1	Logging flag.	Unchanged.
AC2	Not used.	System CCB address.

In addition to the most obvious required input (pathname of file to open), a flag word is supplied to tell RESLV.P whether or not it should check for C2 Logging. If the flag is set to 0, no logging will be done; a negative value will direct RESLV.P to check for "full" C2 logging, and a positive value will indicate "complete" C2 logging. IOPEN.P always sets the logging flag to 0. The system will save the returned system CCB address, depending on which file is opened. For example, when :PROC:PIF is opened, its CCB address is stored in the global location PIFCID.W. When :NET is opened, its CCB address is stored in NETCID.W. This is how AOS/VS system shutdown code accesses these CCBs in order to close the files. The calling sequence from BRKFL.P to IOPEN.P is the following:

```
        XWLDA   0,NMPTR.W,3     ;Byte ptr to filename on stack
        LJSR    BCREATE.P       ;Create the breakfile
          WBR   BERR            ;Bad create

        WSUB    1,1             ;No checks for logging
        LJSR    IOPEN.P         ;Open breakfile
          WBR   BERR1           ;Bad open
        XWSTA   0,CCB.W,3       ;Save CCB on stack for later
                                ;(it goes in PTBL extender)
```

```
#define CKCOMLOG 02
#define CKPRCLOG 010
#define IN_SYSTEM_SPACE 020000000000
```

```
IOPEN.P (*pathname, logging_flag, *CCB_addr_out)
{
/*************************************************************
 * This is an internal open; allocate system memory for a    *
 * system CCB.  Note that although this is a "system CCB," the*
 * system owns it as if it were a user.  That is why its addr *
 * will not be found in the FCB, and will have no use count.  *
 * GOPEN.P would call OFAULT.P to get a user CCB.             *
 *************************************************************/

   CCB_addr = RSMEM (CCBLT);

/*************************************************************
 * Init the CCB: locked and read system buffer command.      *
 * Set the switches for RESLV.P call.  IOPEN.P always        *
 * indicates no log checking and search rules apply.         *
 * Then call RESLV.P to find the file.                       *
 *************************************************************/

   set_bit (CCB_addr, BCBLK);            /* Init CCB locked   */
   CCB_addr->CBFLG = CBSYB;              /* Read system buff  */

   if (logging_flag < 0)
      reslv_switches = CKCOMLOG;          /* Complete logging  */
   else if (logging_flag > 0)
      reslv_switches = CKPRCLOG;          /* Partial logging   */
   else
      reslv_switches = 0;                 /* No logging        */

   call RESLV.P (pathname, reslv_switches, IN_SYSTEM_SPACE,
                 &FIB_IDP, &FNB_IDP, &PCCB_addr);

/*************************************************************
 * If FNB_IDP is 0, only a prefix was input to RESLV.P and   *
 * since prefixes only point to dirs already open on system  *
 * CCBs, the CCB address of the DIR is returned in PCCB_addr. *
 *************************************************************/

   if (FNB_IDP == 0) /* Pathname was just a prefix */
      {
      call CULCK (PCCB_addr):              /* Unlock new CCB   */
      pref_CCB_addr = PCCB_addr;           /* Get "real" CCB   */
      PCCB_addr = PCCB_addr->CBPCB.W;      /* Get "real" parent*/
      if (PCCB_addr != 0)                  /* If prefix not :  */
         call CLOCK (PCCB_addr);              /* get CCB lock  */
      FCB_addr = pref_CCB_addr->CBFCB.W;   /* Save FCB addr    */
      }

/*************************************************************
 * File to open not just a prefix.                           *
 * Got some file info. Get more.                             *
 * Read the FIB.  If the file type is either generic or IPC, *
 * only GOPEN.P can perform the open, so return an error.    *
 *************************************************************/
```

```
   else
      {
      call RAID (FIB_IDP, DEFIB, PCCB_addr,
                                 &FIB_BH_addr, &FIB_addr);

      if ((FIB_addr->FITYP == IPC_TYPE_FILE) ||
          (FIB_addr->FITYP == ?FGFN))
         {                                  /* Before return, */
         call RELB (FIB_BH_addr);           /* release FIB BH,*/
         call KCCB.P (CCB_addr);            /* destroy CCB.   */
         return (Illegal_file_type);        /* OK to return.  */
         }

   /*************************************************************
    * Have FFCB either create (if file not open) or return (if *
    * file already open) an FCB address.  If it is exclusively *
    * open (BFBEO) or opened shared protected (BFBPF), cannot  *
    * open it for this caller.                                  *
    *************************************************************/

      call FFCB (PCCB_addr, FIB_IDP, FIB_addr, &FCB_addr);

      if (check_bit(FCB_addr, BFBEO) ||  /* Exclusive open or */
          check_bit(FCB_addr, BFBPF))    /* prot shared open? */
         {                                  /* Yes!           */
         call RELB (FIB_BH_addr);           /* Release FIB BH.*/
         call KCCB.P (CCB_addr);            /* Destroy CCB.   */
         return (File_exclusively_opened);  /* Can return now.*/
         }
      }

/*************************************************************
 * Init the new CCB.                                         *
 * Bump the file open count in the FCB as well.              *
 *************************************************************/

   CCB_addr->CBFCB   = FCB_addr;          /* FCB addr       */
   CCB_addr->CBUID   = FCB_addr->FBUID;   /* FCB unique ID  */
   CCB_addr->CBPCB.W = PCCB_addr;         /* Parent CCB addr*/
   CCB_addr->CBFIB   = FIB_IDP;           /* FIB pointer    */
   CCB_addr->CBIBN   = -1;                /* No index levels*/
   CCB_addr->CBPTA.W = *CC.W->CPTAD.W     /* PTBL addr      */

   FCB_addr->FBOPN  += 1;                 /* BUMP OPEN CNT! */

/*************************************************************
 * The parent CCB addr is 0 if the pathname was just prefix : *
 * The root has no parent dir.  IOPEN.P will not run into this*
 * condition.  So, bump the parent's use count (one more son) *
 * and unlock it.  If this is the first open, the FIB was     *
 * modified (in FFCB) when the FCB addr was stored at FIFCB.W,*
 * so it must be released MODIFIED (RELM).                    *
 * Note: this is IOPEN.P, so FBSCB.W will always be 0 because *
 * IOPEN.P does not open directories (except for :SWAP and    *
 * :PAGE at system init which are closed immediately after).  *
 *************************************************************/
```

```
   if (PCCB_addr != 0)
      {
      PCCB_addr->CBUSC += 1;              /* Bump parent CBUSC */
      call CULCK (PCCB_addr);             /* and let it go.    */

      if (FCB_addr->FBSCB.W != 0) || (FCB_addr->FBOPN == 1)
         call RELB (FIB_BH_addr);         /* FIB not modified  */
      else
         call RELM (FIB_BH_addr);         /* FIB modified      */
      }

/*************************************************************
 * Establish the caller's access privileges to the file.     *
 * ESTAC.P stores them in the CCB.  However, users can only  *
 * open dirs and use the returned channel number as inputs to *
 * system calls, such as ?GNFN.  User I/O to directory type  *
 * files is a restriction.  So, once it has been established *
 * that the user is privileged to open the dir, give him NO  *
 * more access rights, causing any attempt to do I/O to fail. *
 *************************************************************/

   call ESTAC.P (PCCB_addr, &ACL_privs);     /* ACL in CCB    */
   if (error) return (error);                /* ERFAD, maybe? */

   if (FCB_addr->FBTYP == DIRECTORY_TYPE)    /* Users cannot  */
      CCB_addr->CBSTS [ACL_bits] = 0;        /* do I/O to dirs*/

   if (FCB_addr->FBTYP == UNIT_TYPE)         /* If unit, open  */
      call UNIT.P;                           /* unit specially */
   else
      if (ACL_privs & write_access)          /* Else set true  */
         set_bit (CCB_addr, BCBFWE);         /* write acc bit  */

/*************************************************************
 * Finally, no more errors possible.                         *
 * Return CCB addr to caller and unlock it.                  *
 * GOPEN.P would have to UNPIN the CCB page here, but this is *
 * IOPEN.P and the CCB is in ring 0.                         *
 *************************************************************/

   *CCB_addr_out = CCB_addr;
   clear_bit (CCB_addr, BCBLK);
   return;

} /* end of IOPEN.P */
```

3.4.4 File Close Services

File Close Services perform the function of closing a file. Naturally, it performs operations exactly opposite to those of File Open Services. The open CCB is always destroyed, whether the request is from the user or the system. If the file open count reaches zero on a file close, the Funny FIB in the FCB is flushed to disk (back to the FIB) and the FCB is destroyed. The heart of Close Services, KFCB.P and KCCB.P, perform the fundamental close operations.

File Close Services provides corresponding variants to those of File Open Services. They are the following:

Variant	Function
GCLOSE.P	Close a file for a user.
SCLOSE.P	Close a shared file for a user.
RCLOSE.P	Special AGENT close of a user file.
ICLOSE.P	Internal system close of all user channels of a process when the process terms.
ECLOSE.P	Internal system close of a file open on a system CCB.

External requesting modules of GCLOSE.P

Caller	Effect
User	?CLOSE/?GCLOSE - close a file. The AGENT converts normal ?CLOSEs to ?GCLOSEs.

External requesting modules of SCLOSE.P

Caller	Effect
User	?CLOSE/?SCLOSE - close a file. The AGENT converts ?CLOSEs of shared files to ?SCLOSEs.

External requesting modules of ICLOSE.P

Caller	Effect
CLAUC.P	Close all user channels (called by RELMF.P) in process termination code when cleaning up a process' address space.

External requesting modules of ECLOSE.P

Caller	Effect
SINIT1	Close :SWAP and :PAGE directories after IOPEN.P was done to delete all the files within them.
SDOWN.P	Close :PROC:PIF and :PROC:HIF files at system shutdown.
RELMF.P	Close program files (.PR) for all rings at process termination.
MIRROR.P	Close each unit type file in the mirrored LDU. Each unit in LDU is opened separately to extract the DIB, then closed and linked into the LDU structure.
ALLSW.P	Close old hot swap file when proc with a swapfiles specified with non-default size.
PRCER.P	Errors on proc: close swap/page files, IPC spool file, and ring 7 .PR file.
PSIBLD.P	Close hot swap/page file pools.
LMCODE.P	Close microcode file.
WRBRK.P	Close breakfile.

ECLOSE.P (instead of ICLOSE.P) is the logical opposite of IOPEN.P, but this should be clear in the above charts. Since ECLOSE.P is an internal call and requires only one input, the internal CCB address of the file to be closed, this service will be illustrated algorithmically. Moreover, ECLOSE.P is too straightforward to be true. Refer to the KFCB.P and KCCB.P operations to logically follow the actual code path.

```
ECLOSE.P (CCB_addr)      /* Input: CCB address; Output: none */
{
/*************************************************************
 * Validate the CCB (CBUID/FBUID match) to make sure it's     *
 * really a CCB and the correct CCB being released.           *
 * Then save the FCB address for later and release the system *
 * CCB's memory.                                              *
 *************************************************************/

   call VALCID (CCB_addr);

   FCB_addr = CCB_addr->CBFCB.W;

   call RSMEM (CCB_addr, CCBLT);
```

```
/*************************************************************
 * KFCB will decrement the file open count and check if      *
 * the file remains open.  If not, the FCB will be destroyed. *
 * Then the rest of the hierarchy above this file will be    *
 * released as well if parent directory use counts reach 0.  *
 * See KFCB.P and KCCB.P now.                                *
 *************************************************************/

   call KFCB.P (MCA_link_num, MCA_DCT_addr, FCB_addr);
   return;

} /* end of ECLOSE.P */
```

3.4.5 Logical Disk I/O Interface Services

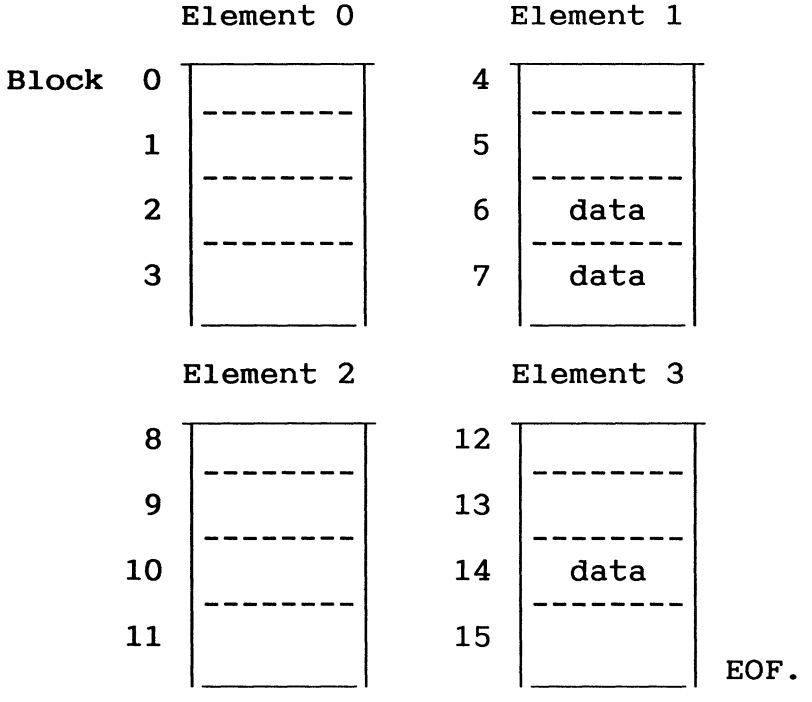
Since a logical disk structure implies a subordinate AOS/VS file structure, logical disk I/O is accomplished by means of file I/O. Therefore, the user is required to first open the desired file and establish a channel number. The channel number along with a packet of information describing the request specifications become the arguments to the kernel implementation of these system calls. The channel number provides access to the CCB, where the request information contained in the packet will be transferred. The File Management system call services that will process this data and enqueue the logical disk request are RDB.P and WRB.P. They are accessible either directly via the system calls ?RDB and ?WRB, or indirectly through the AGENT via the system calls ?READ and ?WRITE.

The essential request information that must be provided by the user includes the first disk block number at which to begin the transfer (?PRNH/?PRNL), the total number of blocks to transfer (?PSTI), the number of bytes in the last disk block to transfer (?PRCL), and the user buffer address to where the actual data will be transferred (?PCAD.W). If the user wants 2049. bytes transferred beginning at block 100. of file FOO, the packet parameters would be initialized as follows:

```
?PRNH = 100.     Begin transfer at block 100.
?PSTI = 5        Transfer 5 disk blocks.
?PRCL = 1        But only one byte in block 5.
```

The ?RDB/?WRB control block does not pend after the logical disk I/O is enqueued. Since the call to NQCCB is the very last operation performed by these services, they are able to omit the call to CWAIT after enqueuing the request because there would be nothing to do (except return) when the request completed. Hence, after the request is made, the ?RDB/?WRB call returns to the system call processor, where the control block is released, and the requesting TCB remains pended. The system call processor knows to do this because RDB.P/WRB.P zeroes out the TCB address in the control block before returning. The CCB post processor unpends the TCB, whose address is stored at offset CBTCB.W of the CCB, when the I/O request is complete. The ?ALLOCATE system call, which enqueues an "allocate disk blocks" CCB request, implements the same mechanism. In all other CCB requests, the CB address is stored at CBTCB.W.

There is another system call that allows users to make logical disk I/O requests. The status word offset in the ?BLKIO packet is used to describe the type of request desired, which can be ?RDB, ?WRB, ?PRDB or ?PWRB. ?BLKIO is functionally equivalent to any of the system calls selected in the packet. However, other special features are available as well. For example, suppose the "Read Next Allocated Element" (RNA) is selected on a ?BLKIO/?RDB request, and the data element of the input starting block is not allocated on disk. This request will begin reading blocks at the next allocated data element until the specified number of blocks are read, a second unallocated data element is encountered, or an end of file condition is reached. If a file is likely to have "holes" (unallocated data elements) in it, such as PRV file created by LINK with a large number of reserved, shared pages, an RNA request could be helpful. The following example illustrates how an RNA request works:



If the first block number to read is 0, and the total number of blocks to read is 15, RNA would begin reading at block 4. This is because blocks 0-3 have never been written to; data element 0 has not been allocated. Since the next element is the second unallocated element encountered, the request terminates after reading the four blocks in data element 1 (two of which have not been allocated, but null blocks are returned). If even one block contained data in element 2, the request would have read all remaining blocks after block 4 and terminated normally. RNA only skips over unallocated data elements that occur prior to the first data transfer; otherwise, the read terminates.

?BLKIO provides two additional features: Modified sector I/O and disk controller status information requests, which are available when used in conjunction with physical I/O.

The RDB.P/WRB.P services are presented in the C-based algorithm below. Data manipulation, validity checking, specific unit type dispatching (RDB.P/WRB.P are interfaces to Unit I/O Management, as well), request enqueuing, and CB/TCB management are illustrated and discussed. The calling process' input accumulators are represented by parameters in the TCB. The inputs and outputs to RDB.P/WRB.P are the following:

Variable	Input	Output
TAC0.W	Not used.	Undefined.
TAC1.W	Channel number.	Number of bytes actually transferred.
TAC2.W	Address of packet.	Unchanged.

```
#define SCDCG 0144

RDB.P/WRB.P (TAC1.W, TAC2.W)
{
   /************************************************************
    * This is a system call with a packet.                     *
    * Get the necessary data into system space.                *
    ************************************************************/

   channel_num = TAC1.W;
   caller_pkt  = TAC2.W
   wblm (caller_pkt, &sys_pkt, pkt_len);

   /************************************************************
    * Find the (user) CCB for the input channel.               *
    * DFAULT.P returns error code n CERWD of control block,    *
    * which the system call processor will interpret.          *
    ************************************************************/

   PTBL_addr = *CC.W->CPTAD.W;
   call DFAULT.P (channel_num, PTBL_addr, &CCB_addr);
      if (error) return;              /* Error code in CERWD */

   /************************************************************
    * Got the CCB.  Do validity checking.                      *
    * All error returns now must unpin the CCB page (which     *
    * DFAULT.P pinned!) and return 0 bytes transferred before  *
    * returning errors.                                        *
    ************************************************************/

   /* Store num blks to xfer.  Error out on null request.     */
   if ((CCB_addr->CBNBK = sys_pkt->PSTI & 0377) == 0)
      {
```

```
   call UNPIN (CCB_addr, PTBL_addr);
   TAC1.W = 0;
   return (Invalid_system_call_parameter);
   }

/* Must be a valid block count.                                    */
if (sys_pkt->PRNH.W & 037740000000 != 0)
   {
   call UNPIN (CCB_addr, PTBL_addr);
   TAC1.W = 0;
   return (Invalid_system_call_parameter);
   }

/* CCB must not be locked already.                                 */
if (get_bit (CCB_addr, BCBLK))
   {
   call UNPIN (CCB_addr, PTBL_addr);
   TAC1.W = 0;
   return (Simultaneous_requests_on_same_channel);
   }

/* Cannot be peripheral type CCB (?FGFN, IPC file)                 */
 if (CCB_addr->CBSTS & BCBPE))
   {
   call UNPIN (CCB_addr, PTBL_addr);
   TAC1.W = 0;
   return (Wrong_I/O_type_for_OPEN_type);
   }

/* CCB/FCB unique IDs must match.  Panic 6034 if not!!!            */
call VALCID (CCB_addr);


/************************************************************
 * CCB/Request A-OK so far.   Init the CCB some more.       *
 ************************************************************/

TCB_addr = *CC.W->CATCB.W                 /* Save TCB addr   */
CCB_addr->CBTCB.W = TCB_addr;             /* TCB addr to CCB */
CCB_addr->CBPTA.W = PTBL_addr;            /* and PTBL addr   */
CCB_addr->CBUAD.W = sys_pkt->PCAD.W       /* and buffer addr */

if (TCB_addr->TSYS.W == ?RDB)             /* If ?RDB, CCB    */
   CCB_addr->CBFLG = CBRED;               /* read command;   */
else                                      /* if ?WRB, CCB    */
   CCB_addr->CBFLG = CBWRI;               /* write command.  */

/************************************************************
 * If this is a unit CCB, let the correct unit I/O handler *
 * finish the request.  The unit type is stored in bits    *
 * 11-15 of the CCB status word, and is used as an index   *
 * in the LDSP (long dispatch) instruction.                *
 ************************************************************/

if (CCB_addr->CBSTS & BCBUN)
   {
   dispatch_to_unit_handler (CCB_addr->CBSTS & CUMSK);
   }
```

```
/*************************************************************
 * Check requestor's access to the file.                     *
 * User opens of dir files were given NO access privileges   *
 * when the file was opened.  Only AOS/VS, specifically,     *
 * Resolution Services, can perform directory file I/O.      *
 * Return accurate error messages on access denial.          *
 *************************************************************/

if (TCB_addr->TSYS.W = ?RDB)              /* ?RDB ?          */
   if (! CCB_addr->CBSTS & read_access) /* Yes, read accs? */
      {                                   /* Nope ...        */
      if (CCB_addr->CBFCB.W->FBTYP = DIRECTORY_TYPE)
         return (Illegal_file_type);      /* Dir error mess  */
      else
         return (Read_access_denied);     /* Other err mess  */
      }

if (TCB_addr->TSYS.W = ?WRB)              /* ?WDB ?          */
   if (! CCB_addr->CBSTS & write_access)/* Yes, writ accs? */
      {                                   /* Nope ...        */
      if (CCB_addr->CBFCB.W->FBTYP = DIRECTORY_TYPE)
         return (Illegal_file_type);      /* Dir error mess  */
      else
         return (Write_access_denied);    /* Other err mess  */
      }

/*************************************************************
 * Access allowed.  Store disk block number and last block   *
 * byte count in the CCB.                                    *
 *************************************************************/

CCB_addr->CBDBH.W = sys_pkt->PRNH.W | (sys_pkt->PRCL << 23.);

/*************************************************************
 * DMTST: verifies all referenced user I/O pages valid,      *
 *        and if so, faults and pins them.                   *
 * CHRG:  charges caller for I/O and checks if time slice    *
 *        will be up during this call.                       *
 *************************************************************/

call DMTST (CCB_addr->CBNBK * 512., CCB_addr);
call CHRG (SCDCG, CCB_addr->CBNBK, CCB_addr);

/*************************************************************
 * Finally, ENQUEUE the CCB request!!!                       *
 * Then, since there is no more to do here, there is no      *
 * point in keeping the CB around.  Tell the system call     *
 * processor the CB can be freed. The post-processor, PPCUS  *
 * will unpend the TCB, unpin user buffer pages, and return  *
 * any data to the caller's packet when the I/O is           *
 * complete.                                                 *
 *************************************************************/

call NQCCB (PPCUS, CCB_addr);             /* Enqueue request */

*CC.W->CATCB.W = 0;                       /* Poof! CB=daemon */
return;                                   /* End of RDB/WRB! */

} /* end of RDB.P/WRB.P */
```

3.5 Shared Protected Files

The shared protected file services provided by File Management work in conjunction with Connection Management to allow a server process to grant customer processes controlled access privileges to a shared file. The user interface to this service is via the ?SOPPF and ?PMTPF system calls. The AOS/VS 7.50 File Management routines that implement this service are SOPPF.P and PMTPF.P.

SOPPF.P is the service called to open a shared protected file. Callers of SOPPF.P may request to be the first shared protected file opener. If the file is already open, the caller will be refused access permission. Otherwise, the file is opened (shared), and a protected file ID is returned. Once the file is opened with ?SOPPF, it can only be opened by subsequent ?SOPPFs; ?SOPPF is effectively the same as an exclusive open unless the first opener grants subsequent openers open privileges.

The first opener explicitly grants subsequent openers the right to open the file via the ?PMTPF (PerMiT Protected File access) system call. ?PMTPF enables the first opener to specify distinct, privileged PIDs permitted to open the file as well as the access rights to be granted. The file access information established by the first file opener for another PID is stored in a Protected File Permission Block (PPB) for that PID. The PPB parameter definitions are the following:

Offset		Protected File Permission Block (PPB)
PPBFW.W	0	PPB forward link.
PPBBK.W	2	PPB backward link.
PPBFO	4	First opener PID (bits 4-15) and Ring (bits 1-3).
PPBPID	5	PID of permittee.
PPBFCB.W	6	FCB address.
PPBR3	10	Permitted access for ring 3.
PPBR4	11	Permitted access for ring 4.
PPBR5	12	Permitted access for ring 5.
PPBR6	13	Permitted access for ring 6.
PPBR7	14	Permitted access for ring 7.
PPBEV	15	Unused.
PPBLN	16	Length of PPB.

Certain FCB fields relate to shared protected files as well. The relevant parameter definitions are the following:

Offset		FCB Fields involving Shared Protected Files
FBST2	46	Shared protected file status word.
FBFOP	47	Pid and Ring of first opener of ?SOPPFed file.
FBPPB.W	50	PPB chain queue descriptor (head).
FBPPBB.W	52	PPB chain queue descriptor (tail).
		:

The PPB establishes the access privileges for a specified process ID and ring. These access privileges will be additional privileges that the subsequent opener may not already possess. The queue descriptor that links the chain of PPBs for a shared protected opened file is FCB_addr->FBPPB.W (head pointer) and FCB_addr->FBPPBB.W (tail pointer). The first opener must issue a separate ?PMTPF for each process/ring tandem for which access rights are being established. Although only one PPB holds the access rights to all rings, individual ?PMTPFs must be issued to set the access rights for each ring.

Before access privileges are given to subsequent openers, PMTPF.P must ensure that the first opener is a server and the specified PID is a customer of that server. PMTPF.P scans the connection table (CNXTB.W) to verify that a connection exists between the caller's PID/ring and the target PID/ring. The Connection Management service CSFIND.P provides this interface. This stumbling block provides yet another measure of security in the protected shared file mechanism. (The exception to this rule is in the case of PMGR.PR, which opens :PMGR:PMGR.SF with ?SOPPF and grants the local PMGR in ring 3 read/write access to the file. Peripheral Manager connections are implicit.)

When the Connection Management validations are complete, the additional access rights to the customer process may be established. PMTPF.P allocates a PPB from the PPB database pool PPBLN.DB (if no PPB exists for the specified PID) and enqueues it to the head of FCB_addr->FBPPB.W. (There is no special FCB lock bit for FBPPB.W because the parent CCB remains locked during the queue instructions.)

The access rights given by the first file opener to each ring of a customer PID are stored in the PPB along with the PID number. PMPTF.P does not permit the caller to issue subsequent openers access privileges other than its own. Therefore, FCB_addr->FBST2, which holds the access rights of the first opener, is "ANDed" with additional access privileges being offered. The bit positions corresponding to access privileges in FCB_addr->FBST2 and PPB_addr->PPBRn are the following:

```
FBPOA (013) - Owner access
FBPWA (014) - Write access
FBPAA (015) - Append access
FBPRA (016) - Read access
FBPEA (017) - Execute access
```

Only the first file opener may issue ?PMTPF. PMTPF.P validates this by comparing the first opener's PID/ring in FCB_addr->FBFOP with the caller's PID and ring. If a process other than the file first opener attempts to issue ?PMTPF, an "Invalid protected file ID" error is returned.

The first file opener may issue subsequent ?PMTPF calls to modify privileges granted to a specific PID/ring. This may be done to either add or revoke access privileges. If, as a result, the access privileges for all rings are null, PMTPF.P deallocates the PPB for that PID. Even if the customer has its own ACL access to the file, the initial ?PMTPF must grant non-null access privileges in order to allow the customer to open the file.

Now, when subsequent processes attempt to ?SOPPF a file, the following conditions must hold true:

1) a valid protected file ID is passed;
2) a PPB exists for the calling PID; and,
3) the calling PID/ring has access to the file.

SOPPF.P first checks the input protected file ID. When the first opener opened the file, SOPPF.P did not generate a random protected file ID, but wittingly assigned it the address of the file's FCB. For subsequent openers SOPPF.P scans each FCB page for an FCB address that matches the protected file ID. If a match is found and the FCB address is valid (it is marked in use in the FCB page header bit map word, CMBMW.W, and there is a first opener), the protected file ID is assumed to be correct and valid. This FCB address is saved at CCB_addr->CBFCB.W. This approach has been implemented because there is no other way of verifying that the input protected file ID indeed points to an FCB.

Next, SOPPF.P scans the PPB chain on FCB_addr->FBPPB.W for the caller's PID. Then, the access privileges for the caller's current ring are retrieved from the PPB, which replace the caller's original access privileges in CCB_addr->CBSTS. The result must give at least read or write access; otherwise, "File access denied" results. Because the newly granted access privileges replace the caller's original access privileges, rather than supplement, not even superusers may access a file being managed with ?SOPPF/?PMTPF.

Grants of access to subsequent openers can be revoked by the following first opener actions: breaking the connection, either explicitly (?DRCON) or implicitly (?RESIGN as a server), or by terminating the customer process (?CTERM). The PPB corresponding to the affected PID is dequeued from FCB_addr->FBPPB.W and deallocated, effectively revoking access. When the subsequent opener closes the file, further attempts to reopen it will be unsuccessful. If the first opener closes the file or terminates, all PPBs on FCB_addr->FBPPB.W will be deallocated. No further ?SOPPFs are possible until the last subsequent opener closes the file. This is because the first opener opens the file exclusively.

The shared protected file feature under AOS/VS 7.50 provides a measure of file access security. A server process can regulate exactly which processes are permitted access to a file. For example, the Peripheral Manager (PMGR) issues ?SOPPF to open :PMGR:PMGR.SF at global PMGR initialization. The PMGR issues ?PMTPF to grant read/write access to the local PMGR in ring 3 of all processes at process initialization. The local PMGR makes a request to the global PMGR, who in turn issues the ?PMTPF and returns the protected file ID (via an IPC). This mechanism makes it impossible for any user process to open :PMGR:PMGR.SF without permission from the PMGR.

3.6 Access Control Privileges

Associated with all AOS/VS file types (except link files) is an Access Control List (ACL). The ACL is used to define users' access rights to files. The ACL consists of a string of usernames and the access control privileges prescribed to each. The string of usernames may contain valid templates (e.g., +, -, *). A typical ACL looks like this:

```
MATT,OWARE X.PUB, $+, +,RE
```

The string portion of the ACL is stored on disk in the File Access Control (FAC) block. The pointer to the FAC is found at FIB_addr->FIFAC. The universal ACL is the access that is permitted to all usernames. That is, any privileges that are common among all names in the ACL string, including the "+" template, form the universal ACL. If there are no common privileges, there is no universal ACL. The universal ACL is stored in bits 11.-15. of FIB_addr->FISTS. Consider the following examples:

```
ACL:              MATT,OWARE X.PUB, $+, +,RE
FIB_addr->FIFAC: MATT,OWARE X.PUB, $+, +,RE
FIB_addr->FISTS: <NULL>

ACL:              MO,OWARE CURLY,WARE LARRY,ARE +,RE
FIB_addr->FIFAC: MO,OWARE CURLY,WARE LARRY,ARE
FIB_addr->FISTS: RE

ACL:              +,OWARE
FIB_addr->FIFAC:
FIB_addr->FISTS: OWARE
```

There may be a slight performance improvement if only a universal ACL is established because further disk I/O to read the FAC may be eliminated. However, Directory Management realizes this possibility and therefore tries to allocate the FAC in the same disk block as the FIB. Establishing only a universal ACL does economize disk space, although a miniscule amount.

When a process accesses a disk file for any reason, the access control privileges for the calling process' username must first be determined. Directory Management isolates the username from the process table extender (PXTN_addr->PUNM) and reads in the FAC (if it exists). The ACL in the FAC is examined from left to right for a match with the username. Upon encountering a match, the access control privileges assigned to the username (and/or template) in the ACL become the access rights of the calling process to the file. If there is no match in the string, the universal ACL privileges are assigned to the process. If there is no universal ACL, the calling process is denied file access. The access control privileges are stored in the CCB at CCB_addr->CBSTS. Virtually all user file access requires the creation of either a user or a system CCB.

AOS/VS 7.50 has provided the following ACL-related services:

1) SACL.P - sets the ACL to a file (?SACL).
2) ISACL.P - AOS/VS internal ?SACL
3) DOACL.P - called from CREATE.P, sets the ACL to a file.
4) GACL.P - returns the ACL of a file (?GACL).

5) ESTAC.P - sets a user's access control privileges to a specified file.

6) PESTAC.P - returns a user's access control privileges to a specified file.

SACL.P writes the input ACL of the specified file to the FIB and FAC (if a string is present in the ACL). If the calling process has write access to the file's parent directory, the caller may modify the ACL in any of the directory's subordinate files (at that hierarchical level). If the calling process does not have write access to the directory, it must have owner access to the specific file being referenced. GACL.P retrieves the ACL of the specified file by reading in the FIB and the FAC (if it exists).

ESTAC.P is called by all File System subcomponents to set the access control privileges for a process (user) which attempts to access a file. The access control privileges are set in the file's CCB (CCB_addr->CBSTS). Once the privileges are stored in memory, they are accessed via the CCB until the file is closed. PESTAC.P is also called by the File System to retrieve the access control privileges for a process (user). The returned privileges are tested for the type of access being requested. Below are the bit positions in CCB_addr->CBSTS that represent the AOS/VS 7.50 access control privileges:

APOWN (013) - Owner access
APWRT (014) - Write access
APAPN (015) - Append access
APRED (016) - Read access
APEXC (017) - Execute access

The ACL management of LDU type files functions slightly differently from other file types. The ACL of an LDU is specified by the system manager during a DFMTR session. An ACL block is allocated in the visible space of the LDU. When the disk is initialized, the ACL block is read to retrieve the ACL. VCREATE.P, called by LDU Initialization to graft the LDU into the existing directory hierarchy, allocates a FAC to which the ACL is written. The ACL of LDUs may be changed while they are initialized, which causes the FAC to be modified. However, when the LDU is released, the FAC is not written back to the ACL block. Therefore, when the LDU is re-initialized, it retains the original ACL specified in the ACL block.

The exception to this rule is the system root directory. Its ACL is always "+,E" despite what is found in its ACL block. Furthermore, its ACL may never be changed.

3.7 C2 Logging

C2 security class event logging is a function of Host Management. However, C2 events are found in many different operating system components. The AOS/VS 7.50 File System events that constitute security-relevant information necessary to obtain a C2 rating, as defined by the Department of Defense Trusted Computer System Evaluation Criteria, December 1985, are the following:

C2 Security Class File System Events

Code	Symbol	Event
920	LOPENCODE	File open
922	LCLOSCODE	File close
924	LDELPCODE	File delete (by pathname)
925	LRUDACODE	File UDA read (by pathname)
926	LWUDACODE	File UDA write (by pathname)
927	LCUDACODE	File UDA create (by pathname)
928	LRELPCODE	Logical disk unit release
929	LCREPCODE	File create (by pathname)
931	LDELCCODE	File delete (by channel)
932	LUDARCODE	File UDA read (by channel)
933	LUDAWCODE	File UDA write (by channel)
934	LUDACCODE	File UDA create (by channel)
937	LINITCODE	Logical disk unit initialization
938	LRENPCODE	File rename (by pathname)
939	LACLPCODE	File ACL change (by pathname)
942	LRENCCODE	File rename (by channel)
943	LACLCCODE	File ACL change (by channel)
945	LSOPNCODE	Shared prot file, first open
946	LSSOPCODE	Shared prot file, subsequent open
947	LPACCCODE	Permit access to shared prot file

4 CCB Request Management

4.1 Overview

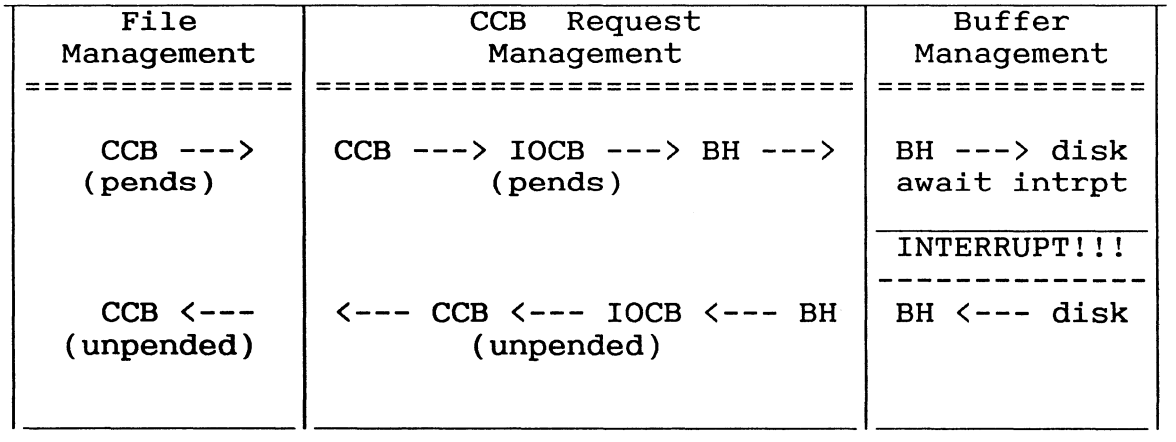
CCB Request Management is the intermediary file system component between File Management and Buffer Management. File Management initiates CCB requests. Request-specific information is gathered from the caller and stored in the CCB, which is "enqueued" by File Management to CCB Request Management. CCB Request Management grabs the CCB and ultimately becomes responsible for setting up a buffer header that contains the logical disk address of the data: essential information that the controller needs in order to execute the data transfer. The complicated and tedious procedure of calculating the logical disk address of the data involves traversing the file's index level structure. Finally, the buffer header is passed to Buffer Management where the physical unit(s) of the LDU containing the data is determined, and the request is enqueued to the appropriate Unit Definition Block (UDB).

Calculating the logical disk address is not a trivial task. The logical disk address of a data block in files with zero levels of indexing is simply the starting file address (found in the FIB) plus the relative data block number. If the file has two levels of indexing, the high index block must be read, the correct offset into it must be calculated, the low level index block must be read, the offset into it must be calculated, and finally the data element must be read. Index block I/O alone can lead to an enormous amount of preliminary I/O before the actual data transfer is enqueued. CCB Request Management implements an index level optimization scheme, which facilitates the finding of index blocks and decreases the number of total requests to the controller. This optimization alleviates what could be a heavy performance hit to the entire system. Both CCB and IOCB parameters are utilized to hold index level traversal data during processing.

All CCB requests are passed to CCB Request Management. However, actual disk data is not always passed back. Read, write, and read-system-buffer requests always involve an explicit data transfer. Allocate blocks, delete file and truncate file requests to modify the file structure in accordance with the request. However, the caller is not expecting any data in return. In all cases, an IOCB is necessary to enqueue disk I/O.

In summary, CCB Request Management interacts with File Management and Buffer Management in servicing logical disk requests. Logical disk I/O requestors specify the file block number and the number of total blocks to transfer. File Management stores this in the CCB. A CCB Request (NQCCB) is made to CCB Request Management, and the requestor's CB (or TCB) is pended. CCB Request Management converts the input CCB request specifications to a logical disk address, allocates an IOCB to store index level information used in determining the logical disk address of the data, and enqueues a buffer header

to Buffer Management. Buffer Management calculates the physical disk address of the disk location of the data and enqueues the buffer header(s) to the appropriate unit. Finally, the disk driver takes control, and the physical transfer will take place. When the disk controller interrupts the host, the data transfer will be complete. The buffer header post-processor unpends the waiting IOCB, which in turn decides whether to unpend the requestor's CB (or TCB) or to enqueue another buffer header. The following diagram summarizes the relationships among the components involved with CCB Request Management.



4.2 I/O Control Block (IOCB)

4.2.1 Definition

The most common use of the term "control block" pertains to the database that saves the state information of a code path that may pend. For example, a system control block (CB) saves process information during system call processing. Similarly, an I/O control block (IOCB) is merely a "control block" that saves the data that CCB Request Management uses when calculating the logical disk address of an I/O request. Since the code path pends at least once during CCB request processing, this object is necessary.

4.2.2 IOCB Scheduling

IOCB pending implies that IOCBs are schedulable entities, which indeed is the case. File Management provides the NQCCB service, which allocates an IOCB and enqueues it to the active IOCB queue, IORUN.W, in the order of the requesting process' priority enqueue factor. The Disk Manager Task, a system task which is always found at the top of ELQUE, searches for ready IOCBs on IORUN.W and runs (or resumes) each I/O request. Therefore, following the IOCB enqueue, NQCCB must ready the Disk Manager Task control block and force a reschedule, so that the I/O request will be serviced "instantaneously." The Disk Manager Task CB is defined at the global label DMTSK and begins execution at location RUNLC1. The routine (called by NQCCB) to "wake up" the Disk Manager Task is called DWAKE.

```
DWAKE                                   /* No inputs, no outputs */
{
/*************************************************************
 * Check the status of the Disk Manager Task control block.  *
 * If the "NOT READY TO RUN" bit is set, the DMTSK is        *
 * currently not running, so clear the bit, which will make  *
 * it ready to run.  Also set the reschedule bit in the mother*
 * processor PPCB, which is the only JP that runs DMTSK.     *
 *************************************************************/

   call XLOCK (DMTSK, BPTRAN);          /* Get DMTSK trans lock */

   if (check_bit (DMTSK, BPSRY))        /* DMTSK ready to run?  */
   {                                    /* NOT YET!             */
      clear_bit (DMTSK, BPSRY);         /* Make it ready to run */
      set_bit   (MPPCB.W, BPRSCH);      /* Force reschedule on  */
   }                                    /* mother JP only       */
```

```
/*************************************************************
 * Indicate that the Disk Manager Task was "woken up" by     *
 * setting DMFLG.  This flag is checked by the DM Task just  *
 * before it pends itself to see if DWAKE was called while it *
 * was already active.  The DM Task will not exit if the     *
 * flag becomes set, but will restart at the top of its      *
 * scheduling loop (and will find something to do).          *
 *************************************************************/

   DMFLG = -1;                      /* Indicate the DWAKE called */
   clear_bit (DMTSK, BPTRAN);       /* Release transition lock   */
   return;                          /* Next guy to be rescheduled*/
                                    /* on mother proc is DM Task */
} /* end of DWAKE */
```

AOS/VS 7.50 imposes the restriction that only the mother JP enqueue buffer headers to I/O controllers. Furthermore, only the mother processor handles I/O interrupts. These restrictions apply to all devices, including logical disks, physical disks, unit devices and user defined devices. Such file system restrictions presently exist because AOS/VS was originally designed to be a uniprocessor operating system. Multiprocessor support requires extensive utilization of locks (critical regions) and full concurrency control. Furthermore, the reliability of the AOS/VS 7.50 file system is extremely high. At this time, the present file system's strength, stability, and reliability are not sacrificed.

Since the Disk Manager Task branches to the CCB Request Management code that ultimately enqueues a buffer header to the appropriate logical disk unit, it must already be running on the mother processor. It would be unacceptably inefficient to allow the Disk Manager Task to run on an arbitrary JP, have the buffer header enqueued to the buffer header wait queue BHWQ.W, and wake up yet another special task on the mother to enqueue the buffer header.

Several system components that do not necessarily run on the mother processor attempt to enqueue a buffer header directly to a UDB (e.g., flushing a modified shared page to disk). Therefore, there must exist both an intermediary mechanism of informing the mother that some daughter JP wants to make a buffer header request, and a system task that actually enqueues the buffer header. The intermediary mechanisms are special, global queue descriptors to which buffer headers are temporarily enqueued before the mother can access them and enqueue them to their respective devices. There are three of these queues in the AOS/VS 7.50 file system:

1) BHWQ.W - Buffer Header Wait Queue.
2) UIOQUE.W - Unit I/O Queue
3) UPPRC.W - Unit I/O Post-Processing Queue

Daughter JPs enqueue disk destined buffer headers to BHWQ.W and unit destined buffer headers to UIOQUE.W. Since a system task that always runs on the mother processor is the Disk Manager Task, it takes responsibility for removing buffer headers from these queues and for enqueuing them where they really belong. After the daughter JP enqueues a buffer header to BHWQ.W or UIOQUE.W, it calls DWAKE to signal that the Disk Manager Task has work to do.

Buffer headers are enqueued to UPPRC.W by the mother processor if the unit I/O post-processor is called from interrupt level. In order to simplify MP memory management, unit I/O post processing is done at base level in AOS/VS 7.50. Since the Disk Manager Task runs at base level, this scheme has been implemented.

The general functions of the Disk Manager Task have now been presented. Here they are summarized in the order they are executed by the Disk Manager Task.

RUNLC: (loop of the following)

(1) Dequeue unit I/O buffer headers from UIOQUE.W and enqueue them to the correct unit device.

(2) Dequeue unit I/O buffer headers from UPPRC.W and dispatch to the correct unit post processor.

(3) Dequeue disk I/O buffer headers from BHWQ.W and call NQBHR to enqueue them to the correct UDB.

(4) Search IORUN.W from head to tail for "READY TO RUN" IOCBs and run them. Control is transferred to CCB request processing code via an XJMP instruction. When an IOCB pends or the request completes, control is transferred back to the top of the RUNLC loop in the same way. The majority of CCB Request Management concerns the running of IOCBs.

(5) If a call to DWAKE comes through during this session, repeat this session (steps 1-4). Otherwise, pend and jump back to the top of the AOS/VS scheduler (SMON0).

DWAKE always sets a global flag, DMFLG, to -1. The Disk Manager Task always clears this flag at the top of the RUNLC loop. The DM Task rechecks the flag just before it pends. If the flag was set to -1, it indicates that DWAKE was called while the DM Task was executing. There was either a buffer header enqueued to one of the intermediary queues or an IOCB enqueued to IORUN.W.

Whatever the case, the DM Task must then jump to the top of the RUNLC loop and repeat its entire procedure, because the request may or may not already have been serviced. For example, if the DMTSK was only up to step (2) when File CCB Request Management enqueued a new IOCB and set DMFLG to -1, the IOCB would be run on the first pass. However, the DMTSK had no way of knowing who called DWAKE or for what reason, so it must make another pass regardless of when DMFLG is set. If the flag is still 0 just before the DM Task is about to pend, CCB Request Management services were not requested. The DM Task can set the "NOT READY TO RUN" bit and transfer control back to the top of the scheduler.

4.2.3 IOCB Processing: Flow of Control

When an IOCB is run for the first time, control is transferred to the starting PC, which NQCCB initializes in the IOCB. The IOCB remains active (on IORUN.W) until the caller's request is complete. A very high-level view of the systematic steps of CCB Request Management (IOCB Processing), to be discussed later in detail, is outlined below.

I. Disk Manager Task
   A. Select IOCB from IORUN.W
   B. Run IOCB

II. CCB Request Pre-Processing
   A. Initialize IOCB given request specifications in CCB.
   B. Command Dispatch
      1. Read Blocks
      2. Write Blocks
      3. Allocate Blocks
      4. Read System Buffer
      5. Delete File
      6. Truncate file (part 1 and 2)

III. CCB Command Processing
   A. Check EOF Considerations
   B. Index level optimization work
   C. Index level traversal
   D. Enqueue buffer header(s) for disk I/O
   E. Pend awaiting disk I/O completion

IV. CCB Request Post-Processing
   A. Unpend requestor's CB or TCB
   B. Deallocate IOCB
   C. Done, jump back to Disk Manager Task

It is important to note that "CCB Request Management" refers to this layer of the AOS/VS File System component. CCB Request Pre-Processing, CCB Command Processing, and CCB Request Post Processing each refer to individual sections within CCB Request Management. The above outline will be followed as closely as possible throughout the course of this chapter.

4.2.4 IOCB Parameter Definitions

The following illustration of IOCB offsets and detailed descriptions of their significance will explain a great number of previously unexplained CCB Request Management concepts and design theory.

Offset		I/O Control Block (IOCB)
IOFWD.W	0	IORUN.W Forward link.
IOBAK.W	2	IORUN.W Backward link.
IOCCB.W	4	Requestor's CCB address.
IOSTW.W	6	IOCB status or wait key when IOCB pended.
IOSPC.W	10	Saved PC.
IOSL0.W	12	Return address of save level 0.
IOSL1.W	14	Return address of save level 1.
IOSL2.W	16	Return address of save level 2.
IOSL3.W	20	Return address of save level 3.
IOSL4.W	22	Return address of save level 4.
IOSBH.W	24	Start of IOCB self-contained Buffer Header ...
IOSTK	24	Start of IOCB DELETE pseudo-frame (see 4.6.4)
IOICB.W	24	The address of this IOCB.
IOBFL.W	24	Buffer LRU Queue Descriptor (head pointer).
IOBBL.W	26	Buffer LRU Queue Descriptor (tail pointer).
IOFFL.W	30	FCB Buffer List Queue Descriptor (head ptr).
IOFBL.W	32	FCB Buffer List Queue Descriptor (tail ptr).
IOQLK.W	34	Driver enqueue link word.
IOADR.W	36	User buffer address to which data is transferred.
IOST	40	IOCB BH status word.
IONBL	41	Number of blocks to transfer.
IOMAP.W	42	Process table address of requestor.
IOUPD.W	44	IOCB BH post-processor address.
IODAH.W	46	Logical disk address of data.
IOPPL.W	50	UPPRC.W link (not used).
IODDP.W	51	IOCB Delete Data Block pseudo-frame pointer. (see Section 4.6.4)
		... End self-contained IOCB Buffer Header
IOTPC	60	Num of index levels left to traverse (negated).
IOREH.W	61	Offset into data element of the data.
IOQHI.W	63	Data element number of the data.
IOERR	65	IOCB error word.
IOIXH.W	66	Indexing (double)word.
IONLV	70	IOCB Flags and number of index levels.
IODEH.W	71	Data element size of file.
IODT1.W	73	IOCB temporary variable 1.
IODT2.W	75	IOCB temporary variable 2.
IOLBC	77	Last block correction count.
IORBC.W	100	Running byte count.
IOFCB.W	102	FCB address of requestor.
IOBFA.W	104	Buffer header address of read system buffer BH.
IOPRIO	106	Priority of this CCB request.
IOTBK	107	Number of blocks to transfer (same as IONBL).
IODSA.W	110	Logical disk address of data (same as IODAH.W).
IOLTH	112	Length of IOCB.

4.2.5 IOCB Static Parameters

NQCCB is the service that provides the interface between the File Management layer and the CCB Request Management layer in the logical disk I/O request procedure. NQCCB allocates an IOCB and initializes it with certain static data, relevant to the calling process, that the IOCB will need to reference during the course of CCB command processing. It also enqueues the IOCB to IORUN.W. The requestor's CCB address (offset IOCCB.W) is acquired during CCB Request Pre-Processing to extract the request-specific information as defined by the user. It is accessed during other phases of CCB Request Management whenever the CCB needs to be referenced or modified, which is especially common during index level traversal. NQCCB was careful to convert the logical CCB address to a physical address, since the referenced ring 1 user CCB would probably not be mapped. (DFAULT pinned it, so it is sure to be in memory!)

NQCCB stores the logical ring 0 FCB address (offset IOFCB.W), since the FCB I/O-in-progress bit must be set when the IOCB request "offically" begins (when DM Task schedules the IOCB to run). The caller's process table is saved (offset IOMAP.W) for use by the disk driver when assigning map slots for the physical data transfer (routine SWAMP).

4.2.6 IOCB Pending Mechanism (and Associated Parameters)

The IOCB status word indicates to the Disk Manager Task that the IOCB is either ready to run or pended. Before the IOCB is enqueued to IORUN.W, NQCCB initializes its status word (offset IOSTW.W) to "READY TO RUN". The Disk Manager Task will search for this status when deciding which IOCBs to run. Any other status means that the IOCB is pended awaiting some event. The following constitute valid IOCB status values:

```
IORDY (1) = IOCB ready to run
IOBWT (2) = IOCB waiting for any buffer header
BH_addr   = IOCB waiting on a specific buffer header
FCB_addr  = IOCB waiting on FCB I/O-in-progress completion
IOBMW (3) = IOCB waiting on the Bit Map FCB Global lock
IOWWT (5) = IOCB waiting on the Bit Map FCB Withdraw lock
```

When CCB Request Management must assign a system buffer header either for a read system buffer request or to read in a file index block and there are no buffers available, the IOCB must pend. It uses the IOBWT status as a pend key. When any buffer header is freed, Buffer Management will first attempt to assign it to any waiting IOCBs (before base level waiters).

When CCB Request Management enqueues a buffer header, it must await I/O completion. But, the buffer header post-processor must know which IOCB to unpend when the I/O completes. CCB Request Management stores the specific buffer header address in IOSTW.W so Buffer Management will be able to unpend the correct IOCB.

The FCB address is used as a pend key when the IOCB must pend during CCB Request Pre-Processing if file I/O is already in progress. This means that if there is already another request in progress on the same file at (or beyond) the CCB Request Management layer, the IOCB must pend awaiting its predecessor's completion.

The Bit Map FCB locks are needed to ensure that only one IOCB allocates (withdraw from the LDU bit map) or deallocates (deposit to the LDU bit map) disk blocks at a time. If the IOCB cannot get the Global Bit Map FCB lock to acquire exclusive access to the Bit Map FCB, it sets IOBMW and pends. If the IOCB cannot get the Withdraw lock, which must be set to withdraw disk blocks, it sets IOWWT and pends. The Global lock is acquired for disk block deposits and the Withdraw lock is acquired for disk block withdraws.

Now is an opportune time to introduce the IOCB pending mechanism. IOCBs do not pend as control blocks or task control blocks pend. A special pending implementation was designed because IOCBs are not system tasks scheduled independently by the AOS/VS main scheduler. They have neither a mapped context nor use of a stack. This implies that all "stack" data be saved within the IOCB, including return addresses of subroutine calls. When an IOCB must pend to await disk I/O completion, a free buffer header, or one of the FCB Bit Map locks, the status word is filled in with the appropriate "pend key," and the address to resume processing is stored in the IOCB (offset IOSPC.W - "saved PC"). Then, since CCB Request Management is temporarily finished with this IOCB, it jumps back to the RUNLC loop of the Disk Manager Task to allow continued processing of IOCBs on the queue. When the IOCB is unpended (a match occurs when the unpend routine checks IOSTW.W), the status is changed to IORDY and DWAKE is called to summon the Disk Manager Task, which will resume processing the IOCB.

NQCCB intializes the "saved PC" IOSPC.W to the starting address of IOCB processing represented by the label RUNRD. When the Disk Manager Task "runs" an IOCB, it actually does an XJMP indirect through offset IOSPC.W of the IOCB!

```
    XJMP    @IOSPC.W,3   ;Run or resume IOCB processing.
```

It makes no difference whether the IOCB is running for the first time or whether it was just unpended and is resuming its processing; the PC is always saved in IOSPC.W. The Disk Manager Task is only concerned about IOCBs with a "READY TO RUN" status.

Due to the reasons mentioned, the system stack is off limits to all CCB Request Management routines that may pend. CCB Request Management easily compensates for stack data storage by allocating a large database, the IOCB, for storage of the pertinent information that needs to be saved throughout the request. However, since WSSVR instructions cannot be used by any subroutines that may pend, an alternative method must be devised. The IOCB reserves five save levels for return addresses. Each subroutine, instead of issuing a WSSVR, saves the return address from AC3 into one of the save levels in the IOCB. Each subroutine must always use the same save level in order for there to be consistency within all of CCB Request Management code. Furthermore, no subroutine with designated save level n may call a subroutine which uses the same save level. Storage of return addresses into static save levels is hard-coded into each routine. The save level offsets in the IOCB and the subroutines that use them are defined as follows:

Save Level	CCB Request Management Subroutines that use save levels
IOSL0.W	RDDEL, WRDEL, RDSYB, DELFIL, TRNCF1, TRNCF2, IFWAIT
IOSL1.W	CLDSK, INDEX
IOSL2.W	CLREM, CLDEL, CLRELE, MBLKN, LBLKN
IOSL3.W	IASBU, LASBU, IBLKW, IBWAIT
IOSL4.W	GROW, GROFL, RNA

An example of how RDDEL (Read Data Element operation) interfaces with subroutine INDEX (traverse file's index levels) is

```
RDDEL:   WINC    3,3              ;Assume normal return
         XWSTA   3,IOSL0.W,2      ;Save return address
                 :
                 :
         XJSR    INDEX            ;Loop through index levels
           WBR   .IDXER           ;Error
                 :
                 :

INDEX:   WADI    2,3              ;Assume normal return
         XWSTA   3,IOSL1.W,2      ;Save return address
                 :
                 :
         XJMP    @IOSL1.W,2       ;Return
```

4.2.7 IOCB Dynamic Request-Specific Parameters

Before CCB Request Management can begin processing the CCB command, the request specifications are moved from the CCB to the IOCB. These are the first operations done by CCB Request Management before command processing begins. The data element size is extracted from the FCB and moved to IOCB offset IODEH.W. The starting block number of the data specified in CBDBH.W is translated into data element number and offset into the data element. The conversion is done by dividing the starting block number by the elementsize:

```
data_element_number = CCB_addr->CBDBH.W / IOCB_addr->IODEH.W
offset_into_element = CCB_addr->CBDBH.W % IOCB_addr->IODEH.W
```

The quotient is the data element number, which is stored at IOCB offset IOQHI.W. The remainder is the block offset into the data element, which is stored at IOCB offset IOREH.W. The offset appellations refer directly to the mathematical division: "Quotient (HIgh)" and "REmainder (High)", respectively. CCB Request Management uses this information to determine how many buffer headers will need to be enqueued to service the request. This is because A SEPARATE BUFFER HEADER MUST BE ENQUEUED FOR EACH DATA ELEMENT! Disk blocks within data elements are logically contiguous, and Buffer Management transfers only logically contiguous blocks per single buffer header request. Logically contiguous data elements within a file do not necessarily have logically contiguous disk addresses; therefore, another buffer header must be enqueued. (Buffer Management may break up CCB Request Management's buffer header request into several requests for reasons such as encountering remapped bad blocks during the transfer, the data element spanning a unit boundary, and lack of a sufficient number of map slots to complete the transfer in one shot.)

Another piece of data that will be used during command processing while index level traversal is taking place is the current number of index levels at the start of the request, saved in bits 8-15 of IOCB offset IONLV. Index level optimization routines and EOF extension routines need to check and possibly modify this field. One complementary field, IOTPC, contains a count of the number of index levels left to be traversed. Offset IOIXH.W contains the offset into each of the index blocks in the following format:

```
Bits 0-7   = 0
Bits 8-15  = offset into highest index block (denoted X1)
Bits 16-23 = offset into middle  index block (denoted X2)
Bits 24-31 = offset into lowest  index block (denoted X3)
```

In addition to holding the number of index levels in the file, offset IONVL contains two flag bits:

```
IORNA (0)  = Phase I/II RNA Request Bit
IOTRAN (1) = IOCB Transition Lock Bit
```

Before an RNA (Read Next Allocated element) request actually reads a block of data, it is said to be in Phase I. The request remains in Phase I until the first file block is transferred. The block can either be the first block requested or the first block of the next allocated element following the requested (unallocated) block. After the first block is transferred, CCB Request Management sets bit IORNA to indicate the RNA request has entered Phase II. This means that the next time an unallocated data element is encountered before the request is complete, the request will be terminated prematurely. The IOCB transition lock is defined in AOS/VS 7.50, but not used.

When the caller of BLKIN or RAID requests that a specific block be read into a system buffer, CCB Request Management must assign a system buffer, which contains a predefined ring 0 data address. The buffer header address is temporarily stored in IOCB offset IOBFA.W and later passed back to the caller, which is responsible for releasing it. In this way the caller can modify the file by releasing the buffer and flushing its modified contents to disk.

User read/write request data transfers are done directly to/from the physical memory page(s) containing the specified data address. Nevertheless, Buffer Management must have a buffer header to enqueue to the device's UDB, where the disk driver world will find it and effectuate the physical data transfer. A system buffer header cannot be temporarily assigned and released by CCB Request Management for a number of reasons. First of all, if a system buffer header were assigned for every user request, the buffer pool may be exhausted too often, causing too many IOCBs to be pended and a very slow system. Secondly, it would introduce an intermediate level in the procedure: disk -> system -> user. The data would have to be copied from system space to user space explicitly. Thirdly, user requests often specify more than one block of data. A system buffer header points to an area of only one block. A huge number of buffer headers might have to be enqueued to satisfy the request. CCB Request processing must be fast, so the data must be transferred directly from the disk to the user page. This leaves a very significant question unanswered: Where does the buffer header come from?

A section of the IOCB itself is defined as the buffer header for user read/write requests. The IOCB buffer header begins at offset IOSBH.W. The offsets following IOSBH.W must exactly match the corresponding system buffer header offsets. CCB Request Management will pass the address of IOCB_addr->IOSBH.W to Buffer Management, which will assume it is just an ordinary buffer header. Even though CCB Request Management indexes it with IOCB parameters (e.g., IOADR.W), Buffer Management indexes it with buffer header parameters (e.g., BQADR.W). A list of CCB commands and the buffer headers each enqueues to Buffer Management to satisfy the request should be of benefit. Note that ALL commands use system buffer headers to read in index blocks, but this usage is not listed.

CCB Command	Buffer Header Usage
CBRED	IOCB Buffer Header. Caller either user via NQCCB or system via NQCRQ.
CBWRI	IOCB Buffer Header. Caller either user via NQCCB or system via NQCRQ.
CBALL	IOCB Buffer Header. Caller always user.
CBSYB	System Buffer Header. Caller always system via BLKIN/CBLKIN.
CBDEL CBTRN1 CBTRN2	No buffer header as no data transfer is requested.

The IOCB buffer header is initialized and updated during the command processing phase before being passed to Buffer Management. For example, as described above, the number of blocks that can be transferred with one buffer header request must be less than or equal to the file's elementsize. If the user's request crosses an element boundary, CCB Request Management must break down the request into several buffer header requests. The number of blocks to transfer on any individual buffer header request is stored at IOCB offset IONBL. This value must be copied to offset IOTBK as well, because Buffer Management destroys IONBL, which is part of the IOCB buffer header. If blocks then remain to be transferred, the number of remaining blocks is stored at CCB offset CBNBK. When CBNBK reaches 0, CCB Request Management knows the request is complete.

Consider a write request of four blocks to file FOO. The request is to file blocks 2-5, and FOO has a data elementsize of 4. Two buffer header requests must be made to Buffer Management because blocks 2 and 3 are in data element 0, and blocks 4 and 5 are in data element 1.

```
Request Specifications: IOCB_addr->IONBL = 4
                         CCB_addr->CBNBK = 4

First  Buffer Header:   IOCB_addr->IODAH.W = (lda of block 2)
                        IOCB_addr->IONBL = 2
                         CCB_addr->CBNBK = 2

Second Buffer Header:   IOCB_addr->IODAH.W = (lda of block 4)
                        IOCB_addr->IONBL = 2
                         CCB_addr->CBNBK = 0
```

The logical disk address of the first block to be transferred must be calculated as well. It is stored at IOCB offset IODAH.W. This is done by traversing the file's index structure until the logical disk address is found in the correct index block. For unshared read and truncate requests, the file block number of the data must occur within the file, or an EOF error will be returned. Shared read, write, and allocate requests will grow the file before transferring data if the specified block lies after the current EOF. The logical disk address to begin file deletion is the file's first address, obtained from the FIB. The logical disk address must be copied to offset IODSA.W as well, because Buffer Management destroys IODAH.W, which is part of the IOCB buffer header. Index level optimization and traversal will be discussed in Section 4.5.

The user data address is stored at IOCB offset IOADR.W. The disk driver will transfer the data to this address. File Management (RDB.P/WRB.P) pinned these pages to eliminate the possibility of the page getting stolen and the data channel transfer destroying the contents of the wrong page. BQADR.W in the system buffer header is set up at system initialization.

Since the disk controller must know whether to read or write data, the CCB command is stored in the IOCB buffer header at offset IOST. This is the buffer header status word, which includes the command type. Bits QTIO (Vanilla I/O) and QTNAB (not a system buffer) are always set for read, write and allocate requests. See Buffer Management for the rest of the status word definitions.

The IOCB buffer header offsets must map to the system buffer offsets, but some may not be pertinent to the IOCB. For example, only system buffers are enqueued to the buffer LRU and the FCB buffer list queues. Buffer header queue links must be defined in the IOCB even though they are not used. The buffer LRU forward link pointer is, therefore, redefined to contain the address of the IOCB. The redefinition parameter is IOICB.W. The system buffer header also defines a corresponding field, BQIOC.W, for this handshaking purpose. The IOCB buffer header post processor checks this field in order to unpend the IOCB awaiting I/O completion. This post-processing routine is RESTR, which is stored by CCB Request Management at IOCB offset IOUPD.W. Another label, RSTRB, is defined at the same address as RESTR, and is referenced as the read system buffer command post-processor. When the disk controller interrupts the host, signalling request completion, RESTR is called from the disk driver interrupt service routine, stored in *IOCB_addr->IOUPD.W. The IOCB address is obtained from the IOCB buffer header, and the IOCB status is made ready. The post-processor wakes up (unpends) the Disk Manager Task so that IOCB processing will continue. It is important to note that this procedure is performed at interrupt level, so there will not be a reschedule until the interrupt is dismissed.

When user read/write requests complete, CCB Request Post-Processing calls the CCB Post-Processor, which must return to the user task the actual number of bytes transferred. The count of the number of bytes transferred is incremented each time a buffer header request completes. This running byte count is stored at IOCB offset IORBC.W. If an entire block was not transferred, IORBC.W rounds up the number of bytes to the next block multiple anyway. This poses no problem for unshared read requests, which must always read a fixed number of blocks, and the whole request must fall within the file. However, shared read and write requests may reference blocks outside the file's EOF. CCB Request Management will grow the file and satisfy the request without an error.

The number of bytes in the last block NOT transferred must be saved, since the entire last block may not have transferred. This last block byte correction is saved at IOCB offset IOLBC.W. The CCB Post-Processor subtracts IOLBC.W from IORBC.W to render the number of bytes actually transferred. The final value is returned to the user in TAC1.W.

If an error occurs during IOCB processing, the error code is temporarily stored at offset IOERR until it can be moved over to the CCB where the caller will be able to act upon it.

4.2.8 IOCB Global Locations

Global Location	Significance
IOCB.DB	IOCB Database Pool
IORUN.W	IOCB Scheduling Queue head pointer
IOTAIL.W	IOCB Scheduling Queue tail pointer
IOCBLK.W	IORUN.W VSMP Generic Queue Lock Word
IOCBP.W	Pointer to active IOCB
IOAC	Number of IOCBs currently on IORUN.W
IOMX	Max number of IOCBs on IORUN.W since boot

4.3 CCB Request Pre-Processing

After an IOCB is allocated and initialized with static parameters by NQCCB, the Disk Manager Task schedules the IOCB and execution begins at address RUNRD. CCB request Pre-Processing consists of initializing some IOCB parameters and setting up the request specifications as described in Section 4.2.7. The parameters which must be initialized are:

```
IOCB_addr->IOERR   = 0      No error yet.
IOCB_addr->IORBC.W = 0      No running byte count yet.
IOCB_addr->IOLBC   = 0      No last block correction yet.

IOCB_addr->IODEH.W = data element size.
IOCB_addr->IONVL   = number of index levels.
IOCB_addr->IOQHI.W = beginning data element number.
IOCB_addr->IOREH.W = block offset into data element.

IOCB_addr->IONBL   = number of blocks in element to read,
                     write, allocate.  Always set to 0
                     for delete and truncate requests.

CCB_addr->CBNBK    = total number of blocks left to read,
                     write, allocate before the request is
                     satisfied.  Delete and truncate do not
                     use this field.
```

These parameters are derived from the input CCB parameters. The data will be converted to a lower level, which Buffer Management and the disk drivers will understand. For instance, Buffer Management only understands logical disk addresses, and drivers only understand physical disk addresses. Finally, the CCB command is extracted, and a call to the command processing routine is performed.

One further test must be done. There must be no other CCB request currently in progress on the same file when a CCB request begins processing. This status is flagged by setting the I/O-in-progress bit in the FCB (FBIOP). If the bit is already set during CCB Request Pre-Processing, the IOCB must pend on this status until some other request completes. The FCB waiter count (FBWTC) is incremented as well. When the request that originally set FBIOP completes, CCB Request Post Processing checks FBWTC and unpends the first IOCB pended on this status.

4.4 File EOF Considerations

CCB Request Command Processing must ascertain the validity of the specified file blocks to be transferred in the request. Unshared read requests may reference data only within the file's boundaries. If the requestor attempts to access a block beyond the file EOF, the request is truncated. Shared read requests may reference unallocated blocks beyond the file EOF. If the requestor has write access to the file, index blocks and the specified data blocks (elements) will be allocated, and the file EOF will be extended. In some cases the number of index levels will grow, which requires extensive file structure modifications and FCB parameter updating. A block of zeroes will be returned to the requestor's buffer. Write requests are similar; the transfer is in the opposite direction.

CCB Request Processing provides routines that check for the above conditions and modify the request specifications and/or file structure accordingly. The four routines are:

a) CEFNU - Check EOF (No Update)
b) CKEOF - Check EOF (and modify it if necessary)
c) GROFL - Grow File (Index Levels only)
d) GROW - Grow File (Index/Data Blocks)

CEFNU is the first subroutine called upon entering CCB read and write command processing. CEFNU truncates non-privileged requests that reference data beyond the file EOF. The purpose of this call is solely to truncate the request length. The criteria for truncation are:

1) Truncate all unshared reads if any of the specified blocks to transfer lie beyond the file EOF.

2) Truncate shared read requests if the requestor does not have write access to the file.

3) Truncate shared read requests and write requests to the maximum possible 32-bit EOF if any of the specified blocks to transfer lie beyond the maximum possible EOF.

The number of blocks to transfer on the current buffer header request, IOCB_addr->IONBL, is decremented so that the last block transferred is the EOF block. The number of blocks left to transfer in the entire request, CCB_addr->CBNBK, is set to zero. The request is effectively complete after the next data transfer. CEFNU does not update the EOF; this is done by CKEOF when the last data transfer completes.

Although read system buffer requests can extend the EOF, CEFNU is not called because only the system issues the CBSYB CCB Command; ring 0 assumes write access.

CKEOF is the complemetary routine to CEFNU. CKEOF is called by those requests that possibly modified the file EOF in order for the EOF extension to be updated in the FCB Funny FIB. CKEOF is called even if a new data element was not added because a new file block could be allocated beyond the EOF, but within the last data element.

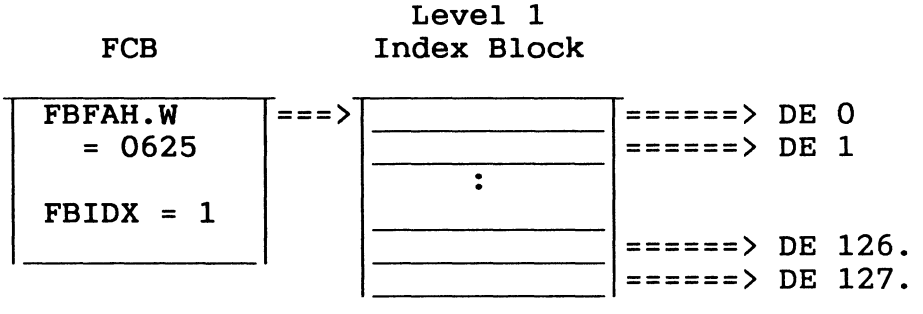
GROFL is the routine that adds index levels to a file if a shared read, write, or read system buffer request will cause the number of index levels to grow. If the condition

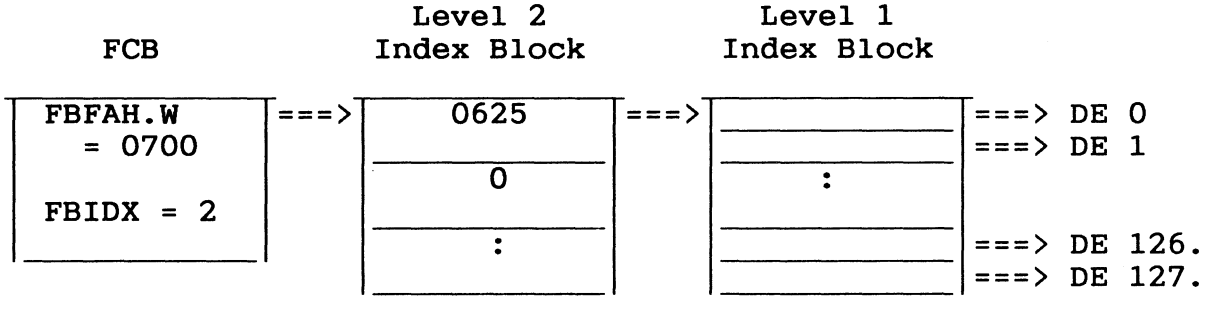
```
Q < 2 ** (7*L), where Q = data element number,
                      L = current index level,
```

holds true, the current number of index levels is sufficient. Otherwise, one or more index levels must be added. For example, a one index level file holds a maximum of

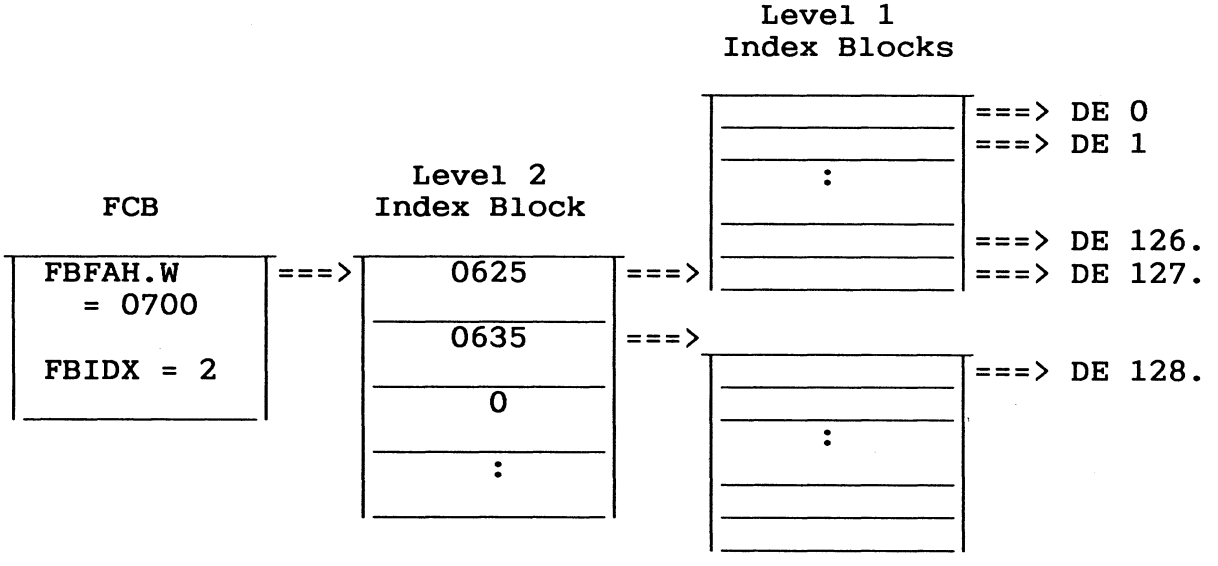
```
2 ** (7 * 1) = 128
```

pointers to data elements. If Q = 129, a two index level file must be built. GROFL calls WDBCK to withdraw one block from the LDU free block pool and converts the block into the high index block of the file, i.e., the file first address (FCB_addr->FBFAH.W). The new index level is reflected in the FCB Funny FIB as well (FCB_addr->FBIDX++). The flush bit is set in the Funny FIB (BFBFL) so this modification will be updated in the FIB when the file is closed. Offset 0 of the new index block is filled with the logical disk address of the old high index block, which is now the low (level 1) block. Below is a graphic representation of what happens when a 1 index level file becomes a 2 index level file.





GROW is a separate routine that allocates data elements as well as index blocks, either before or after the file EOF. GROFL must be called previous to GROW. In the previous example, if a write request to element 128 were made to a 1-index level file, GROFL would first add an index level. Subsequently, GROW would allocate a new level 1 index block whose pointer would be stored at offset 1 of the level 2 index block. GROW would then allocate data element 128 and store its pointer at offset 0 of new level 1 index block. This point is clearly illustrated below.



4.5 File Index Optimization

4.5.1 Methodology

The AOS/VS File System component employs an index optimization scheme in order to reduce the number of total disk I/O requests. The CCB contains the logical disk address of the index block(s) read in the previous request, as well as the offsets into those blocks. The IOCB is initialized to contain the offsets into the one, two, or three level index blocks that the current request will reference. Beginning with the highest index block, if certain current request index block offsets match the previous request's index block offsets, one or two index block reads can be skipped. However, a buffer header must always be enqueued to read the requested data element. This optimization helps to relieve the bottleneck of a "slow disk" when there are many disk requests in the queue.

The CCB fields used in index level optimization are CBIAH.W, CBXIA.W and CBIBN. On each I/O request, CCB Request Management updates these fields to contain the following:

CCB Parameter	1 index level	2 index levels	3 index levels
CBIAH.W	LDA data element	LDA level 1 index block	LDA level 1 index block
CBXIA.W	Not used	Not used	LDA level 2 index block
CBIBN	Offset into level 1 index block	Offset into level 2 index block	Offset into level 3/2 index block

The IOCB fields used in index level optimization are IOIXH.W and IOTPC. IOIXH.W is set up in the following format:

```
Bits 0-7   = 0
Bits 8-15  = offset into level 3 (highest) index block (X1)
Bits 16-23 = offset into level 2 (middle)  index block (X2)
Bits 24-31 = offset into level 1 (lowest)  index block (X3)
```

The (doubleword) offset into the level 3 index block (in the range 0 - 127.) is represented by the symbol X1. The offset into the level 3 index block is represented by the symbol X2. The offset into the level 1 index block is represented by the symbol X3. Therefore, IOIXH.W can be represented as follows:

```
IOIXH.W = <0><0><0><X3>   for a 1-index level file
IOIXH.W = <0><0><X2><X3>  for a 2-index level file
IOIXH.W = <0><X1><X2><X3> for a 3-index level file
```

It is trivial to extract these values from the data element number in IOQHI.W. Each index block contains 128. (2 to the seventh power) pointers to either subordinate index blocks or data elements. Thus, if the data element number is 129. (0201), which can be represented in bit notation as 01 0000001, the offset into the low index block is 1 and the offset into the middle index block is 1.

Consider the following situation in which data block 4 is requested from file FOO, whose elementsize is equal to 4. Assume that FOO is one index level deep and this is the first I/O request to the file on this channel. When processing reaches CCB Request Management, all index optimization offsets will indicate that no previous I/O has yet occurred on the channel (initialized with -1). The one and only high index block will be read in and its buffer header enqueued to the FCB. The offset into it that references the data element (offset 1) is saved at CCB offset CBIBN. Next, the data element will be read in and its logical disk address saved at CCB offset CBIAH.W. Now suppose that the caller makes a subsequent request for data block 5. Since this request will access the same data element as the previous request, the index block need not be read. This fact is determined by comparing values at IOIXH.W and CBIBN which are equal, indicating the same index block offset match. Furthermore, the logical disk address of the data element is already available in the CCB and can be read immediately, without CCB Request Management touching an index block.

Two and three level indexed files utilize CCB parameters to a greater extent to further expedite the data transfer by "skipping over" the same index blocks read in on the previous data transfer. However, the lowest level index block will always be read in (or found on the FCB buffer list queue) in files with index level greater than 1.

Consider the following example. If a read request were made on file FOO (elementsize 4), which referenced block 5120. (element 1280), the high index block would be read in, the offset into it (offset 10.) would be saved at CBIBN, and its logical disk address would be saved at CBIAH.W. A subsequent request to block 5121. would register a match between CBIBN and X2 in IOIXH.W, the offset into the level two index block. The logical disk address of the appropriate level 1 index block is then retrieved from CBIAH.W. Since this is the lowest level index block and the file is greater than one index level deep, the block must be referenced. Thanks to the FCB buffer list, still no disk I/O should take place to read the level 1 index block because it should be found on the queue. The previous request

put it there. The offset into it, which references the correct data element, is extracted from IOIXH.W. The logical disk address of the data element is then retrieved, and the buffer header to read the element is enqueued. This example should prove the great value of both the index level optimization mechanism and the FCB buffer list.

Index level optimization does some statistical record-keeping. The following global counter variables are incremented each time a CBIBN/IOIXH.W match occurs at a specific index level:

X3MAT.W - Number of matches at index level 1 ("X3 match")
X2MAT.W - Number of matches at index level 2 ("X2 match")
X1MAT.W - Number of matches at index level 3 ("X1 match")

4.5.2 Routines

There are three CCB Request Management routines associated with index level optimization. They are called by CCB Command Processing to determine whether or not index levels must be traversed.

PARSQ sets up the IOCB index optimization offsets for the current request and compares them with the CCB offsets relating to the previous request. If a match occurs, the "match return" is taken.

MATCH is called to determine whether the PARSQ match occurred at data level or at index level. If a a data level match occurred, the data element is read in. If a non-data level match occurred (IOTPC != 0), there is still at least one index level that must be traversed.

INDEX is the routine which traverses index levels. Its input argument is the logical disk address of the highest index block at which to begin the downward structural penetration. INDEX returns the logical disk address of the desired data element, which will subsequently be read in. A demonstration of the usage of these routines is exhibited in Section 4.6.1; however, they are sketched out algorithmically below as well.

```
/************************************************************
 * PARSQ: Index Optimization Subroutine.                    *
 * Output: TRUE if a match occurred between the index block *
 * offset referenced in the previous request and that ref'ed *
 * in the current request.  A match means that at least one *
 * index block will not have to be read in and index level  *
 * traversal can be shortened or skipped all together.      *
 ************************************************************/

PARSQ (*match_found);
{
/************************************************************
 * Assume no match will be found for now.                   *
 * Init number of index levels to traverse to current number *
 * of index levels in the file (negation thereof).          *
 * A file not indexed is a file not traversed.              *
 ************************************************************/

   *match_found = FALSE;
   IOCB_addr->IOTPC = -(IOCB_addr->IONLV & 0377)

   if (IOCB_addr->IOTPC == 0)
      return;

/************************************************************
 * Set up IOIXH.W.                                          *
 ************************************************************/

   IOCB_addr->IOIXH.W  = IOCB_addr->IOQHI.W  & 000000000177;
   IOCB_addr->IOIXH.W |= (IOCB_addr->IOQHI.W & 000000037600)<<1;
   IOCB_addr->IOIXH.W |= (IOCB_addr->IOQHI.W & 000007740000)<<2;

/************************************************************
 * One index level optimization.                            *
 * If CBIBN and IOIXH.W match, the data element being ref'ed *
 * on this request is the same as the last request.  There is *
 * no need to read in the index block again.  Increment the  *
 * index level counter IOTPC to indicate one less level of   *
 * indexing necessary.  Return match found true.             *
 * If no match, CBIBN must be updated with THIS request's X3! *
 ************************************************************/

   if (IOCB_addr->IOTPC = -1)                    /* 1 idx lvl? */
      {                                          /*        Yes */
      if (CCB_addr->CBIBN == IOCB_addr->IOIXH.W)/* Elem match?*/
         {                                       /*        Yes */
         X3MAT.W += 1;                           /* Bump global*/
         IOCB_addr->IOTPC += 1;                  /* Got a data */
         *match_found = TRUE;                    /* level match*/
         return;
         }
      else                                       /* No Match!  */
         {
         CCB_addr->CBIBN = IOCB_addr->IOIXH.W;   /* Reset CBIBN*/
         return;                                 /* No matched */
         }
      }
```

```
/************************************************************
 * THREE index level optimization.                          *
 * Three index level must be done before two because we must *
 * begin at the highest index block.                        *
 * First, shift IOIXH.W to get <0><0><X1><X2>.  X3 not needed.*
 * If an X1 match, bump the global count and indicate one less*
 * level of indexing in the traversal.  We cannot return yet *
 * because level 2 must be checked first.  If level 2 finds a *
 * match also, that's two index blocks eliminated!          *
 * If no match, must update CBIBN and take no match return.  *
 ************************************************************/

   IOCB_addr->IOIXH.W >>= 8;
   if (IOCB_addr->IOTPC == -3)
      {
      if ((CCB_addr->CBIBN >> 8) == (IOCB_addr->IOIXH.W >> 8))
         {
         X1MAT.W += 1;                      /* Bump glob count  */
         IOCB_addr->IOTPC += 1;             /* Level 3 done!    */
         x1match = TRUE;                    /* Boolean for later*/
         }
      else
         {                                          /* No Match!  */
         CCB_addr->CBIBN = IOCB_addr->IOIXH.W;      /* Reset CBIBN*/
         return;                                    /* No matched */
         }
      }

/************************************************************
 * Two index level optimization.                            *
 * If CBIBN and IOIXH.W match, X2 (and maybe X1 from above  *
 * as well) is the same for the previous and this request.  *
 * The middle level index block will not be read.  Do the   *
 * accounting and take the match return.                    *
 * If no X2 match, we still might be a 3 level file that had *
 * an X1 match.  If so, take the match return.  If not, take *
 * the no match return.  In either case, CBIBN must be updated*
 * because the X2 offset it now different.                  *
 ************************************************************/

   if (CCB_addr->CBIBN == IOCB_addr->IOIXH.W)   /* X2 match?  */
      {                                         /*        Yes */
      X2MAT.W += 1;                             /* Bump global*/
      IOCB_addr->IOTPC += 1;                    /* To level 1 */
      *match_found = TRUE;                      /* with match!*/
      return;
      }
   else                                     /* NO X2 Match!  */
      {
      CCB_addr->CBIBN = IOCB_addr->IOIXH.W; /* Fix CBIBN     */

      if (x1match)                          /* Prev X2 match?*/
         {                                  /*           Yes */
         *match_found = TRUE;               /* So take match */
         return;                            /* return.       */
         }
      else                                  /* Else, no      */
         return;                            /* match found.  */
      }
 } /* end of PARSQ */
```

```
/*************************************************************
 * MATCH: Determine the match type flagged by PARSQ and      *
 * return the logical disk address of either the data element *
 * to read (if data level match) or the index block at which  *
 * to begin traversal (other match).                          *
 * For clean presentation of this algorithm and its dependency*
 * with RDDEL (Section 4.6.1), also return a flag indicating*
 * whether the match was a data level match.                  *
 *************************************************************/

MATCH (*lda, *at_data_level)
{
/*************************************************************
 * If at data level (no index levels left to traverse),      *
 * return the logical disk address of the data element. IOTPC *
 * is 0 only if the file is one index level deep.             *
 *************************************************************/

   if (IOCB_addr->IOTPC == 0)
      {
      *at_data_level = TRUE;
      *lda = CCB_addr->CBIAH.W;
      return;
      }

/*************************************************************
 * If there was an X2 match (only one level left to traverse) *
 * return the logical disk address of the level one index     *
 * block at which to start traversal.                         *
 *************************************************************/

   if (IOCB_addr->IOTPC == -1)
      {
      *at_data_level = FALSE;
      *lda = CCB_addr->CBIAH.W;
      return;
      }

/*************************************************************
 * There was only an X1 match.                                *
 * Return the logical disk address of the level 2 index block *
 * found at CCB offset CBXIA.W.                               *
 *************************************************************/

   *at_data_level = FALSE;
   *lda = CCB_addr->CBXIA.W;

} /* end of MATCH */

/*************************************************************
 * INDEX: Traverse index levels.                              *
 * Input: Log disk addr of index block at which to start.     *
 * Output: Log disk addr of data element.                     *
 *       : hole = TRUE if a zero is encountered in any index  *
 *         block, signifying a "hole" in the file.  All reqs  *
 *         return nulls, but shared reads reqs and write reqs *
 *         later fill up the hole by allocating new blocks.   *
 *         If hole is true, lda_data_element goes undefined.  *
 *************************************************************/
```

```
INDEX (lda_index_block, *lda_data_element, *hole)
{
/*************************************************************
 * Initialize variables.                                     *
 *************************************************************/

   hole = FALSE;
   next_lda = lda_index_block;

/*************************************************************
 * This loop traverses index levels.                         *
 * It exits when IOTPC reaches 0 (at data level).            *
 * First, get the lda of the current block to read at IODAH.W *
 * where LBLKN needs it.  At first, it is an index block.    *
 * After first loop pass, either data element or index block. *
 *************************************************************/

   while (TRUE)
   {
      IOCB_addr->IODAH.W = next_lda;

    /*********************************************************
     * If at data level, then return the data element lda.   *
     * If the file is 1 or less index levels, store the lda  *
     * of the data element of this transfer in CBIAH.W.      *
     * This may be used next time during optimization.       *
     *********************************************************/

      if (IOCB_addr->IOTPC == 0)
         {
         *lda_data_element = next_lda;
         if ((IOCB_addr->IONLV & 0377) <= 1)
            CCB_addr->CBIAH.W = next_lda;
         return;
         }

    /*********************************************************
     * If at level 1 (IOTPC == -1), save the level 1 index   *
     * block lda in CBIAH.W.  If at level 2 or 3, save the   *
     * level 2 index block in CBXIA.W.                       *
     *********************************************************/

      if (IOCB_addr->IOTPC == -1)
         CCB_addr->CBIAH.W = next_lda;
      else
         CCB_addr->CBXIA.W = next_lda;

    /*********************************************************
     * LBLKN: checks FCB buffer list for index block.  If not *
     * found, searches the LCB cache buffer list.  Both queues*
     * are searched by logical disk address.  If the index   *
     * block is not found, it is read in from disk.  The IOCB *
     * will pend during the I/O.  The pend key is the BH addr.*
     *********************************************************/

      call LBLKN (next_lda, &BH_addr);

    /*********************************************************
     * LDCADR: returns the lda at the proper offset in the   *
     * index block just read in.  The offset is in IOIXH.W.  *
     *********************************************************/
```

```
      call LDCADR (next_lda);

    /*************************************************************
     * If the retrieved lda is 0, there is a hole in the file.*
     * The caller will take care of this case (specific action*
     * depends on the caller).                                  *
     *************************************************************/

      if (next_lda == 0)
         {
         hole = TRUE;
         return;
         }

    /*************************************************************
     * Made it through an index level.                          *
     * Indicate one less to traverse by bumping IOTPC.          *
     * Then release the index block buffer (not modified).      *
     * Next_lda now contains either the data element address    *
     * (which will be returned) or another index block          *
     * address (for which the loop's procedure will repeat).    *
     *************************************************************/

      IOCB_addr->IOTPC += 1;
      call RELB (BH_addr);

   } /* end of while */
} /* end of INDEX */
```

4.6 CCB Request Command Processing

The CCB Request Pre-Processing code branches to the appropriate command processing routine. These routines are discussed in this section. If an error occurs during processing, the CCB error occurred bit (BCBER) is set. The CCB post-processor of user read, write, and allocate requests examines this bit, and if it is set, forces the caller of the CCB request to take the error return. The error code is returned in IOCB offset IOERR. If no error occurs during processing, the CCB post-processor unpends the requestor normally.

4.6.1 CBRED: Read Command Processing

This command may be initiated by the user or by the system. If the user issues a ?RDB system call, File Management translates it into a CCB request with the CBRED CCB command and calls NQCCB to enqueue the request to CCB Request Management. The data will be transferred to the user's buffer. If the system wishes to issue a read, the two global File Management interfaces are BLKIN (reads one block into a system buffer) and NQCRQ (reads one or more blocks into a ring 0 memory area). ?READ causes a CCB request if the AGENT does not find the requested data in an AGENT buffer. ?SPAGE may defer I/O until a page fault demands it.

There are two types of read requests: shared and unshared. The only difference between unshared and shared reads at the CCB Request Management level is that unshared read requests cannot read data beyond the file's EOF; shared read requests will add index levels to the file (if necessary), allocate the requested data element (and any index structure necessary to logically access it), and extend the file's EOF to satisfy the request.

CCB command processing of all read requests includes traversing the file's index level structure until arriving at the data element number specified in IOQHI.W. The index level optimization algorithm is applied to avoid unnecessary reads of index blocks. But, since the IOCB buffer header is used for user requests, and the IOCB is deallocated when the request is complete, these buffer headers are never enqueued to the FCB buffer list; data elements for user requests are always read in from disk. Only index blocks and file blocks requested by the read system buffer command are enqueued to the FCB buffer list.

After a data element is successfully read, the number of blocks left to read (CCB_addr->CBNBK) is checked. If there are blocks left to read, the IOCB and CCB are updated and another element is read. Remember, one buffer header reads no more than one data element! When there are no more blocks left to read, the request will be satisfied. CCB Request Post Processing will call the CCB post-processor to unpend the pended requestor and the IOCB will be deallocated (or assigned to a waiting CCB).

The following algorithm outlines the CCB Request Management read request procedure. It may look complex at first glance, but the comments are helpful. The algorithm is useful for three reasons. First, it exhibits the actual internal implementation of what happens when a data element is read. Second, it introduces the EOF management and index level optimization routines, which were discussed in Section 4.5. Third, the comments should provide better insight into how reads work at this low level, i.e., exactly when and how an EOF error is detected and returned to the caller.

```
RDDEL: /* READ A DATA ELEMENT */

/*************************************************************
 * CENFU: Checks if request must be truncated.               *
 * Always truncates request if beyond max 32-bit EOF.        *
 * If unshared read OR shared read with no write access,     *
 * truncates request if beyond EOF.                          *
 *************************************************************/

    call CENFU;                          /* See Section 4.4.   */

/*************************************************************
 * Null requests do nothing, so just a return is in order.   *
 * CENFU truncated unshared read requests beginning beyond   *
 * the file's EOF to null requests, with IOERR set to EREOF. *
 * Same with shared read requests with no write access, if   *
 * beyond EOF.  CCB Request Post Processing will move the    *
 * error from IOERR in the IOCB to CBERR in the CCB.         *
 *************************************************************/

    if (IOCB_addr->IONBL == 0)
        goto CCB Request Post Processing;
```

```
/*************************************************************
 * If request is shared read, the file may need to grow.     *
 * GROFL: Adds index levels to the file if 1) the request is *
 * beyond current EOF, 2) more index levels will be needed,  *
 * and 3) the requestor has write access to the file.        *
 * GROFL allocates only one new index block per level grown! *
 * See Section 4.4.                                          *
 *************************************************************/

     if (shared read)
        call GROFL;

/*************************************************************
 * PARSQ: Index level optimization routine.  Sets IOCB/CCB   *
 * indexing parameters in accordance with this request.      *
 * If this request is close enough to the previous one such  *
 * that one or more index block reads may be omitted, "match"*
 * will be set to true.  (MATCH implies a match of IOCB index*
 * optimization offsets with current request specifications.)*
 * If no match, either index levels must be traversed or the *
 * file exists with no data (it is null).                    *
 *************************************************************

     call PARSQ (&match);

     if (! match)
        {
     /*********************************************************
      * If the file is null (true here only if request is     *
      * within the first file element and read is shared),    *
      * first element must be allocated.  WDCBK allocates one *
      * data element and returns its address.  CLDEL zeroes  *
      * it, CKEOF extends the EOF, IMCLR zeroes out the       *
      * waiting user's buffer.  No disk I/O is necessary for  *
      * the transfer.  This data element will be done, so     *
      * UPDATE will determine whether there are more data     *
      * elements to be read.                                  *
      *********************************************************

       if (FCB_addr->FBFAH.W == 0)
          {
          call WDCBK (&FCB_addr->FBFAH.W);
          call CLDEL;
          call CKEOF;
          call IMCLR (&CCB_addr->CBUAD.W);
          goto UPDATE;            /* Continue with request! */
          }

     /*********************************************************
      * The file is not null, so index levels must exist and  *
      * at least one index block must be read in (or found on *
      * the FCB buffer list).  In other words, we are NOT at  *
      * data level and must traverse the index structure to   *
      * find the logical disk address of the desired element. *
      *********************************************************/

       else
          at_data_level = FALSE;
```

```
/*************************************************************
 * If there was a match, either one index blocks has to be   *
 * read in, OR we are at data level.                         *
 * MATCH: returns lda of highest index block to read OR      *
 *        returns lda of data element to read.               *
 *        If the lda is that of an element, at_data_level is *
 *        set true.                                          *
 *************************************************************/

    if (match)
       call MATCH (&log_disk_addr, &at_data_level);

/*************************************************************
 * If we are not at data level, INDEX will traverse the index *
 * structure and returns the logical disk address of the data *
 * element.  If the data element was not allocated, shared   *
 * reads will allocate it and return zeroes to the requestor; *
 * Unshared reads will just return zeroes.                   *
 *************************************************************/

    if (! at_data_level)
       {
       call INDEX (log_disk_addr, &lda_data_element,
                   &data_element_not_allocated);

       if (data_element_not_allocated)
          {
    /*********************************************************
     * Shared read, data element not allocated.              *
     * Shared reads anticipated possible file growth by      *
     * adding new index levels if they would be needed, but  *
     * no new elements were allocated.  They must be now!    *
     *                                                       *
     * GROW allocates possible index blocks (GROFL only added*
     * a level, maybe) and always the data element, CLDEL    *
     * zeroes it out, CKEOF updates the EOF (CEFNU just      *
     * checked it).  No data transfer is done because we know*
     * the new element is all null, so IMCLR zeroes out the  *
     * waiting user's buffer.                                *
     *********************************************************/

          if (shared read)
             {                        /* Alloc index blks maybe. */
             call GROW;               /* Always alloc element!   */
             call CLDEL;              /* Clear out data element. *
             call CKEOF;              /* Update EOF now.         */

             call IMCLR (&CCB_addr->CBUAD.W); /* 0 call buff */
             goto UPDATE;             /* Continue with request!  */
             }

    /*********************************************************
     * Unshared read, data element not allocated.            *
     * The data element must already be within bounds, or    *
     * an EOF error would have been returned.  Unshared      *
     * reads never allocate new data elements.  If they are  *
     * not allocated already, then zeroes will be returned.  *
     * Again, no disk I/O is necessary.  IMCLR just zeroe    *
     * out the waiting user's buffer.                        *
     *********************************************************/
```

```
            if (unshared read)
               {
               call IMCLR (&CCB_addr->CBUAD.W); /* 0 call buff */
               goto UPDATE;          /* Continue with request!  */
               }
            }

/*************************************************************
 * Ready to read the data element.                           *
 * For shared reads, the EOF must be checked/extended if the  *
 * element was allocated beyond the EOF OR if a new block     *
 * was requested within the last element beyond the EOF.      *
 *************************************************************/

        if (shared read)
           call CKEOF;

/*************************************************************
 * Set up the IOCB self-contained buffer header parameters.   *
 *************************************************************/

        IOCB_addr->IODAH.W = lda_data_element;
        IOCB_addr->IOICB   = IOCB_addr;
        IOCB_addr->IOADR.W = CCB_addr->CBUAD.W;
        IOCB_addr->IOST    = QTIO ! QTNAB;

        if (RNA request)
           IOCB_addr->IONLV = IOCB_addr->IONVL | IORNM;

/*************************************************************
 * Enqueue the (IOCB self-contained) buffer header.           *
 * Setting IOSTW.W to the buffer header address changes the   *
 * IOCB status to "not-ready-to-run".  The contents becomes   *
 * the pend key, which the BH post-processor will check when  *
 * the I/O is complete.                                       *
 * After the NQBHR, control is passed back to the DM Task.    *
 *************************************************************/

        IOCB_addr->IOSPC.W = &UPDATE;        /* Save (return) PC  */
        IOCB_addr->IOSTW.W = IOCB_addr->IOSBH.W; /* Start BH addr*/

        call NQBHR (&RESTR, *LCBP.W, &IOCB_addr->IOSBH.W);

        XJMP RUNLC;                          /* Back to DM Task!  */
```

```
/*************************************************************
 * IOCB BH PP (RESTR) unpended the IOCB!                     *
 * The elem has been transferred.  Update IONBL and CBNBK (as *
 * described in Section 4.2.7).  If there is more to read,   *
 * repeat this procedure.  If the request is complete, CCB   *
 * Request Post Processing will call the CCB post-processor  *
 * to wake up the requestor and the IOCB will be deallocated. *
 *************************************************************/

UPDATE:
     if (CCB_addr->CBNBK != 0)          /* More elems to read?  */
        {                               /* Yes!                 */
        IOCB_addr->IOQHI.W += 1;        /* Next data element    */
        IOCB_addr->IOREH.W  = 0;        /* First block in it.   */

        /* If number of blocks left to read <= elementsize    */
        if (CCB_addr->CBNBK <= IOCB_addr->IODEH.W)
           {
           IOCB_addr->IONBL = CCB_addr->CBNBK; /* This is the */
           CCB_addr->CBNBK  = 0;               /* last reqst. */
           goto RDDEL;                         /* READ ELEM!  */
           }

        /* Else read another whole data element               */
        else
           {
           IOCB_addr->IONBL = IOCB_addr->IODEH.W;
           CCB_addr->CBNBK -= IOCB_addr->IONBL;
           goto RDDEL;
           }
        }

     else                               /* No more elems to read*/
        goto CCB Request Post Processing;       /* Clean up. */


  } /* end of RDDEL */
```

4.6.2 CBWRI: Write Command Processing

Write requests are extremely similar to shared read requests, except that the data transfer takes place in the opposite direction. Only the following differences may be noted:

1) If a non-allocated data element is the target of a write request, the element is explicitly allocated. But, it is zeroed out only if the write request does not cover all blocks within the element. Otherwise, it is not zeroed because the request data will overwrite the entire element. Shared read requests allocate the element, but since no data is to be written, it must always be zeroed out.

2) Write requests always enqueue a buffer header to write to newly allocated data elements. Shared read requests do not enqueue disk I/O to read known nulls from newly allocated elements; instead the caller's buffer is explicitly zeroed out.

3) The status word in the IOCB buffer header (IOST) contains both the vanilla I/O bit (QTIO) and the "not a system buffer" bit (QTNAB), as well as the modified bit which signals a WRITE request (QTMOD).

4) The modified bit is set in the FCB (BFBMD) on any successful write.

4.6.3 CBALL: Allocate Command Processing

When CCB Request Pre-Processing dispatches on the CCB command, allocate requests branch to the same command processing routine as write requests. Since allocate requests really equate to write requests of null data, the destination data elements are merely allocated and cleared. The CCB command word (CBFLG) is checked during processing to determine whether the request is write or allocate, and the appropriate action is then taken.

4.6.4 CBSYB: Read System Buffer Command Processing

Read system buffer requests are made exclusively by the BLKIN/CBLKIN service. The objective of read system buffer command processing is to return to the caller a system buffer containing the specified relative file block. Unlike multiple file block reads, it is possible that absolutely no disk I/O takes place. The buffer may already be enqueued to either the FCB or LCB buffer lists. If it is found on the FCB buffer list, which is searched by relative block number, the buffer's use count (BH_addr->BQUSC) is incremented and the buffer header address is returned to the caller. That is the optimal situation.

If the system buffer is not found on the FCB buffer list, it may be enqueued to the LCB. The LCB cache buffer list is searched by logical disk address. Therefore, the file's index structure is traversed to obtain the requested block's logical disk address, and the LCB cache buffer list is subsequently searched. If the buffer is found on the LCB cache buffer list, it is moved to the FCB buffer list and removed from the Buffer LRU. Its use count is incremented as well. The buffer header address is then returned to the caller. This situation is still preferred over enqueuing yet another disk request to read the block.

The typical situation is that the buffer header is found on neither queue. When a file is first opened, the FCB buffer list is initialized to empty. In this case, a system buffer must be assigned. If there are no buffers available, the IOCB must pend with a status of IOBWT, waiting for any system buffer, until one becomes free. Buffer Management will always assign freed buffer headers to waiting IOCBs before base level waiters. When the IOCB gets a system buffer, it is removed from the Buffer LRU and enqueued to the FCB buffer list! Its use count is incremented as well, until it is released. Finally, the buffer header is enqueued to the device and the data on disk is read into the buffer data address. The system buffer, not the self-contained buffer header in the IOCB, is passed to Buffer Management because CCB Request Post Processing deallocates the IOCB, including the buffer header. The caller (i.e., BLKIN and CBLKIN) is responsible for releasing the system buffer header. This functionality provides a clean method of accessing and modifying single data blocks on disk. For example, Disk Information Blocks, Directory Data Blocks, LDU/directory bit map blocks, and IPC spoolfile blocks are obtained in this way, modified, and released either with a modified or flush status set in the buffer header.

Read system buffer command processing is similar to shared read requests in that the file grows index levels and allocates new data elements if the request is beyond the current EOF. Only the system makes read system buffer requests, and the system always has write access to the file. Hence, the file can "legally" be modified. As with shared read requests, the data element is zeroed on disk. The ring 0 data block identified by the system buffer header is zeroed out as well, instead of performing wasteful and needless disk I/O.

By definition, only one block of one data element is read with the read system buffer command. If the caller specifies more than one block, i.e., IOCB_addr->IONBL != 1 and CCB_addr->CBNBK != 0, the system panics with code 6315. No IOCB/CCB parameter updating is performed since only one buffer header is enqueued to the device. As usual, CCB Request Post-Processing calls the read system buffer command post processor, which wakes up the pended requestor. Since the buffer is dynamically assigned, the post-processor returns the assigned buffer header address to the caller in the CCB (offset CBUAD.W).

4.6.5 CBDEL: Delete File Command Processing

The CCB delete command informs CCB Request Management that all file index blocks and data elements must be deposited to the LDU free disk block pool. After this has occurred, the file will have no contents, so the resulting file first address in the FIB must be zeroed. The index block and data element buffer headers on the LCB cache buffer list must be destroyed as well, since they will never be needed again.

File Deletion Services, which is at the Directory Management layer, enqueues the CBDEL command to CCB Request Management before deleting the file's directory data elements. If the delete request comes through while the file is open, File Deletion Services sets the FCB "delete on last close" bit (FBDLE). File Close Services will call File Deletion Services to "really" delete the file when the file open count reaches 0. If the delete request comes through while the file is not open, File Deletion Services enqueues the delete request to CCB Request Management immediately.

Files with no index levels are easily deleted. Only one data element need be deposited to the LDU free disk block pool. This is accomplished by the LDU Management services DEBLK and DEBKS. The former deposits one block, while the latter deposits multiple blocks. Since the LDU Bit Map maintains block allocation information, LDU Management must read it into a system buffer, modify it to reflect the deallocation of disk blocks, and release it (modified). Furthermore, LDU Management realizes that LDU block withdrawal/deposit requests are made exclusively from CCB Request Management. Therefore, the system stack is not used, and the IOCB pending mechanism is executed (in contrast to base level CB pending).

Files with one or more index levels are deleted recursively. One routine deletes all index blocks and data elements subordinate to its caller. The miraculous routine, DELETE, is first called with the address of the high level index block. It is called recursively at each index level to delete all subordinate disk blocks in the file's structure. As a result of the recursion, the actual disk block deallocation begins at the lowest level data element (the byte EOF) and works upwards to the index block. At the end of this section, the succint but comprehensive algorithm will clearly illustrate this clever implementation.

DELETE is no different from most other CCB Request Management subroutines in that the system stack cannot be utilized. Moreover, because it is recursive, the same save level cannot be used to store the return address. Otherwise it would be overwritten on each successive repetition. The return address and other important delete-related data is stored in a redefined area of the IOCB called the pseudo-stack. Upon each recursive call to DELETE, a pseudo-frame, called a Delete Data Block, is pushed on the pseudo-stack.

The pseudo-frame pointer at IOCB offset IODDP.W points to offset 0 of the Data Delete Block corresponding to the present call to DELETE. The pseudo-stack is redefined over the IOCB buffer header area because no disk I/O requests requiring the use of the IOCB buffer are enqueued during delete request processing. Index blocks are read into system buffers. There is enough space for three pseudo-frames, one for each call to DELETE at each index level. The Data Delete Block is defined below.

Offset		Delete Data Block
DDSPC.W	0	Return address.
DDBFA.W	2	Buffer header address of last index block read.
DDLVL	4	Current index level.
DDTPC	5	Temporary counter.
DDSTP	6	Offset in index block at which to stop deleting.
DDBLT	7	Length of Delete Data Block

Before DELETE is called, the pseudo-frame pointer is initialized to the beginning address of the pseudo-stack (IOCB offset IOSTK). The frame is essentially already pushed when DELETE is called. In fact, one of the arguments to DELETE is the pseudo-frame pointer. DELETE stores the return address at Delete Data Block offset DDSPC.W.

When DELETE must read in an index block, its buffer header address is saved at offset DDBFA.W until it is released. Index blocks are released and deposited into the LDU free disk block pool after all subordinate disk blocks to it have been released and deallocated.

DELETE is called for every index block in the file structure to delete all blocks subordinate to it. If a two index level file is being deleted, the level 2 index block contains a possible 128 pointers to level 1 index blocks. DELETE is called to deallocate the level two index block, and recursively for each level 1 index block pointed to by the parent index block. DELETE remembers which offset of the current index block is being deleted at Delete Data Block offset DDTPC. This counter is initialized to the maximum number of pointers in an index block and is decremented when the index block and all inferior blocks have been deallocated. When DDTPC reaches 0, the current index block can be deallocated itself.

When DELETE realizes that the current index level is 1, i.e., all subordinate blocks are data elements, no more recursive calls to DELETE will be made. DELETE keeps track of the current index level at offset DDLVL.

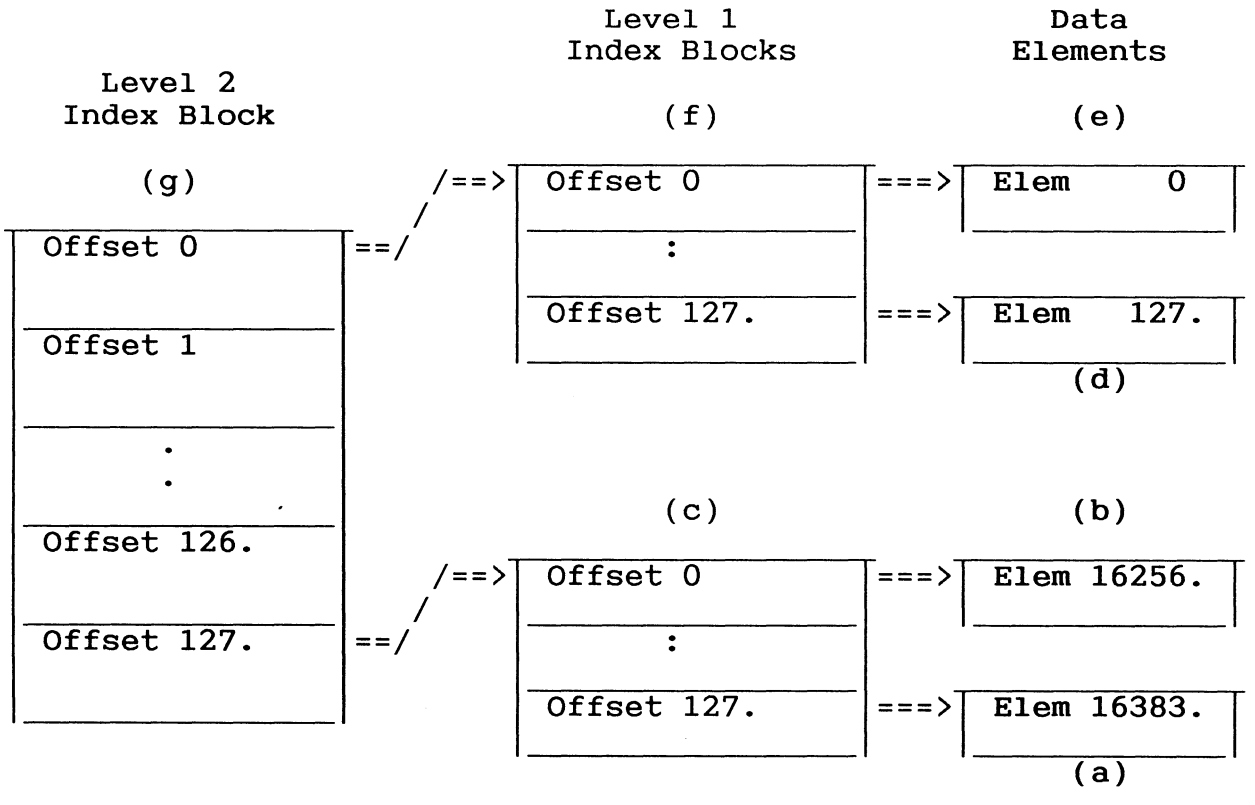
The DELETE routine is used by truncate requests as well as delete requests. After all, a truncate is a de facto delete, only to a lesser extent. Delete Data Block offset DDSTP contains the index level offset at which to stop deallocating inferior file blocks. Only truncate requests utilize this field. Delete requests always slam a -1 here to signify that the whole file is to be deleted.

DELETE is easily explained in theory, but quite involved at the assembly language level. The following high level algorithm should sufficiently illustrate the file block deletion procedure. CCB Delete File Request processing begins at label DELFIL. DELETE is called with the logical disk address of the index block at which to begin deletion. DELFIL calls it with the file first address (if the file is indexed).

```
/************************************************************
 * DELFIL: CCB Delete File Command Processing.              *
 * Delete a file's contents.  This includes all index blocks  *
 * and data elements.                                       *
 ************************************************************/

DELFIL:
{
  /************************************************************
   * If the file is already null, nothing to do.              *
   ************************************************************/

     if (IOCB_addr->IOFCB.W->FBFAH.W == 0)
         goto CCB Request Post Processing;


  /************************************************************
   * If no file indexing, deposit one and only data element.  *
   ************************************************************/

     if (IOCB_addr->IONLV & 0377 == 0)
         {
         call DEBKS (IOCB_addr->IOFCB.W->FBFAH.W);
         goto CCB Request Post Processing;
         }

  /************************************************************
   * File is indexed.  Call DELETE, which will end up         *
   * freeing all index blocks and data elements subordinate   *
   * to the highest index block.  Afterwards, return the high *
   * index block to the LDU free block pool.                  *
   ************************************************************/

     call DELETE (IOCB_addr->IOFCB.W->FBFAH.W);  /* Rel below */
     call DEBLK  (IOCB_addr->IOFCB.W->FBFAH.W);  /* Rel me    */
     goto CCB Request Post Processing;           /* Clean up  */
}
```

```
/*************************************************************
 * DELETE: Return to the LDU free block pool all index blocks *
 * and data elements pointed to by the index block whose      *
 * disk address is passed as an argument.                     *
 * THIS ALGORITHM IS VERY HIGH LEVEL!                         *
 *************************************************************/

DELETE:
{
  /*************************************************************
   * Check each offset in the index block.                     *
   *************************************************************/

     if (DDBLK_addr->DDLVL > 1)
        {
        for (offset = 127; offset > -1; offset--)
           {
         *******************************************************
         * If the index block is level 2 or 3, each non-null  *
         * offset points to a subordinate index block.  Call  *
         * DELETE to deallocate each subordinate index block  *
         * and its subordinate blocks.  Then DEBLK            *
         * deallocates the index block.                       *
         *******************************************************/


           if ( *(index_block_addr + offset*2) != 0 )
             {
             call DELETE (*(index_block_addr + offset*2));
             call DEBLK  (index_block_addr);
             }
           }
        }
     else
         {
         *******************************************************
         * If the index block is level 1, each non-null       *
         * offset points to a data element.  Call DEBKS to    *
         * deallocate each one.  Then DEBLK (of the caller)   *
         * deallocates the index block.                       *
         *******************************************************/

         for (offset = 127; offset > -1; offset--)
           {
           if ( *(index_block_addr + elem*2) != 0 )
             call DEBKS (*(index_block_addr + elem*2));
           }
         }

} /* end of DELETE */
```

Consider the application of CCB Delete File Request Processing to the two index level file below. DELFIL calls DELETE with the address of index block (g). DELETE in turn recursively calls itself for each of the level 1 index blocks, beginning with index block (c). DELETE begins deallocating the blocks pointed to by the index block's last offset, and works upwards until the blocks pointed to by offset 0 are deallocated. Consequently, element (a) would be deallocated first, then all those elements between (a) and (b), and finally element (b). When index block (c) is full of null pointers, it can be deallocated. When all the index and data blocks pointed to by index block (g) are deallocated, DELETE returns to DELFIL, where (g) is finally deallocated. (The order of file block deallocation in the diagram is denoted by the ascending single letter representations.)



4.6.6 CBTRN1: Truncate Command Processing (part 1)

4.6.7 CBTRN2: Truncate Command Processing (part 2)

The CCB Truncate command must be performed in two parts. Base level truncate system call processing enqueues two separate CCB requests to accomplish file truncation. Part 1 processes the bulk of the request by performing the following actions:

1) Clears the remaining data blocks in the same data element following the new EOF,

2) Adds index blocks and a data element if the new EOF is defined where there is a "hole" (unallocated data element) in the file,

3) Deallocates all index blocks and data elements past the new EOF.

Part 2 concludes truncate processing as follows:

1) Checks if file can shrink index levels. If so, the logical disk address at offset 0 of the high index block becomes the first file address. The high index block is deallocated. The process is repeated until the number of index levels is correct.

Separate CCB requests must be enqueued because base level truncate processing must flush the modified index blocks (found on the FCB buffer list) to disk. This must be done to force consistency between the LDU bit map and the actual contents of the index blocks. When both parts of the truncate request are complete, the Funny FIB is flushed to disk to update the new state of the file.

4.7 CCB Request Post-Processing

CCB Request post-processing begins immediately after command processing is complete. The most significant event that CCB Request post-processing initiates is the execution of the CCB post-processor. NQCCB stored the address of the appropriate CCB post-processor in the CCB when the request was enqueued. The five CCB post-processors were outlined in Section 3.3.2. The main objective of all CCB post-processors is to unpend the CB or TCB awaiting request completion. The CCB post-processor returns any data to the user packet as well. Below is the C-based algorithm which outlines the user read, write and allocate CCB post-processor, PPCUS.

```
PPCUS (IOCB_addr, CCB_addr)
{

/*************************************************************
 * Must map TCB to ring 0 to avoid faults.                   *
 *************************************************************/

   call MAPUNW (CCB_addr->CBTCB.W, &TCB_addr);

/*************************************************************
 * If an error occurred during CCB Request processing, the   *
 * error bit is set and the error code is in CBERR.  Decr    *
 * the user's return PC so he takes the error return, and    *
 * stuff him with the error code in AC0.                     *
 *************************************************************/

   if (check_bit (CCB_addr, BCBER))
      {
      TCB_addr->TPC.W -= 1;
      TCB_addr->TAC0.W = CCB_addr->CBERR;
      }

/*************************************************************
 * Always return the number of bytes actually transferred.   *
 * Next, unmap the modified TCB.                             *
 *************************************************************/

   TCB_addr->TAC1.W = IOCB_addr->IORBC.W - IOCB_addr->IOLBC;
   call UNMAP (CCB_addr->CBTCB.W);
```

```
/*************************************************************
 * Hardware I/O bits must be set on the memory pages         *
 * affected by this request because BMC/data channel xfers   *
 * cannot mark the pages.  It is done by the hardware on     *
 * memory reference instructions only.                       *
 *                                                           *
 * MUIOPGR: Set modified and referenced bits.  The memory    *
 *          page was modified and referenced when data was   *
 *          read INTO it FROM disk.                          *
 * MUIOPGW: Propagate modified bit, set referenced bit.      *
 *          The memory page was only referenced when data    *
 *          was read FROM memory TO disk.                    *
 *************************************************************/

   switch (CCB_addr->CBFLG)
      {
      case read_request:
        call MUIOPGR;              /* Mark I/O Page for Read  */
        break;

      case write_request:
         call MUIOPGW;             /* Mark I/O Page for Write */
         break;

     case allocate_request:
         do_nothing();
         break;
      }

/*************************************************************
 * Unpin the data pages pinned for the data transfer.        *
 * This was done by the File Management system call code     *
 * before the CCB request was enqueued.                      *
 *************************************************************/

   if (read_request or write_request)
      {
      offset = 0;
      while (CCB_addr->CBNPG-- > 0)
         {
         call UNPIN (CCB_addr->CBUAD.W + offset); /* Now unpin*/
         offset += 02000;                         /* each page*/
         }
      }

/*************************************************************
 * Unlock and unpin the user CCB.                            *
 *************************************************************/

   call UCULCK (CCB_addr);
   call UNPIN  (CCB_addr);
```

```
/*************************************************************
 * Unpend the TCB by resetting the TCB pend bit.             *
 * (The TCB is already wired when the pend bit is cleared!)  *
 * Then decrement the process' active system call count.     *
 *************************************************************/

   clear_bit (CCB_addr->CBTCB.W->TSTAT.W, ?BTPN);  /* Unpend */
                                                   /*  T C B */
   call DSQCT;                          /* Decr num of syscalls */

 *************************************************************
 * Ready the process table and force a reschedule.           *
 * IWKUP.REL also must determine whether the current JP or   *
 * another will now run the process immediately, or if it    *
 * will remain on the current JP's ELQUE where it is.  If    *
 * this CB/PTBL PNQF is higher than the current element on   *
 * the current JP, reschedule.  Else, check the other JPs to *
 * see if this CB/PTBL PNQF is higher than the current.  If  *
 * so, cross interrupt the JP and force a reschedule.  If    *
 * nothing can be done, just ready the CB/PTBL and keep it   *
 * "in line" on the current processor's ELQUE.               *
 *************************************************************/

   call IWKUP.REL;                      /* Ready process, resch */
   return;                              /* Return to CCB        */
                                        /* Post-processing.     */
} /* end of PPCUS */
```

In addition to calling the CCB post processor, CCB Request Post Processing incorporates some other important functions associated with the completion of the CCB Request. The CCB Request Post Processing functions are listed below in the order that they are executed:

1) Move error code from IOCB (IOERR) to CCB (CBERR).

2) Call CCB Post Processor.

3) Wake up IOCBs waiting on the file. The pend key is the FCB address which CCB Request Pre-Processing stored if I/O was already in progress on the file.

4) Dequeue IOCB from IORUN.W.

5) Either assign the IOCB to the first waiting CCB on CCBWQ.W, initialize it, and enqueue it to IORUN.W; OR return the IOCB to the IOCB database pool (IOCB.DB).

6) Return to the RUNLC loop of the Disk Manager Task.

5 Buffer Management

5.1 Overview

Buffer Management is the File System component that manages the initial allocation, the temporary assignment and release, and the status of system cache buffers. The word cache is usually omitted when referring to system buffers in any other context than when they are enqueued to the LCB Cache Buffer List. A system buffer consists of 512. bytes of system (ring 0) memory, a perfect fit for one disk block of data. Indeed, system buffers are utilized by the File System or external components for the purpose of cleanly retaining in memory the contents of a specific data block of a file. The number of system buffers is a gennable parameter selected when building an AOS/VS system. The "CACHE [128]:" prompt during a VSGEN session refers to the number of system [cache] buffers that system initialization routines will allocate.

Associated with each system buffer is a buffer header. The purpose of the buffer header is to hold low-level, request-specific information, including the logical disk address of the system buffer, the logical disk address of the disk-based data, and the number of blocks to transfer. The buffer header also maintains status information, such as the number of buffer users and buffer waiters.

Buffer headers are ultimately "enqueued" for disk I/O. Before this concept is explained, the reader should be familiar with the LCB and UDB formats, described in LDU Management (Sections 6.2 and 6.3).

In order for a data transfer to take place, Buffer Management must "enqueue" a buffer header to the appropriate physical unit. The unit's buffer header request list is found at UDB offset UDCRQ.W. Buffer Management calls upon the appropriate disk driver to enqueue a buffer header to a specific UDB. The disk driver initiates the data transfer by submitting Programmed I/O commands to the disk controller. Therefore, the UDB ultimately contains the data that the controller uses for the effective physical transfer.

Buffer headers can be assigned by any system component that reads disk-based data whose logical disk address is already known. This is the major difference between why code paths chose to make a direct buffer header request (known logical disk address) or a CCB request (unknown logical disk address). The caller must "assign" the buffer, set up the buffer header, "enqueue" the buffer header (to the desired UDB), pend to await the I/O completion, and finally "release" the buffer. Buffer Management provides services to facilitate this procedure. The following services will be discussed in detail throughout this chapter.

1) ASBUF/BLASB - assign a system buffer

2) NQBHR/NQBH1 - enqueue a system buffer

3) BWAIT - pend awaiting I/O completion

4) RELB/RELM/RELF/RELD - release a system buffer.

5.2 System Buffer Parameter Definitions

Offset		System Buffer Header (BH)
BQBFL.W	0	Buffer LRU (BFLRU.W) Forward Link.
BQBBL.W	2	Buffer LRU (BFLRU.W) Backward Link.
BQFFL.W	4	FCB/LCB Buffer List Forward Link.
BQFBL.W	6	FCB/LCB Buffer List Backward Link.
BQQLK.W	10	Driver enqueue link word.
BQADR.W	12	Ring 0 buffer address to which data transferred.
BQST	14	BH status word.
BQNBK	15	Number of blocks to transfer.
BQMAP.W	16	Process table address of requestor.
BQUPD.W	20	BH post-processor address.
BQLAH.W	22	Logical disk address of data.
BQPPL.W	24	Unit I/O Post-Processing Chain (UPPRC.W) link.
BQDBN	26	Data Block Number.
BQFLGS	27	BH Flags word.
BQUDB.W	30	Unit Definition Block (UDB) address.
BQRBC.W	32	Running Byte Count.
BQSBP.W	34	Status Block Pointer (for physical I/O).
BQFCB.W	36	FCB address (if BH on FCB Buffer List).
BQLCB.W	36	LCB address (if BH on LCB Cache Buffer List).
BQUSC	40	BH Use Count.
BQWTC	41	BH Waiter Count.
BQHLT	42	Length of BH.

5.3 System Buffer Allocation

System buffers and their buffer headers are allocated from GSMEM at system initialization by subroutine BFALL. The booted system file contains the number of buffers to allocate. If the system manager chooses to override the default specs, system initialization prompts for the number of system cache buffers. This number must be greater than the maximum number of Group 1 and Group 2/3 control blocks, 96 in AOS/VS 7.50. The number must not be greater than 1024. System buffer memory is never deallocated.

Newly allocated buffer headers are enqueued to the Buffer Least Recently Used (LRU) chain. The global queue descriptor is:

BFLRU.W - head pointer
BTAIL.W - tail pointer.

The buffer header allocation and initialization consists of the following few steps:

```
BH_addr = GSMEM (BQHLT);            /* Alloc buff hdr mem */
BH_addr->BQADR.W = GSMEM (SCDBS);   /* Alloc buff data mem*/

BH_addr->BQDBN   = -1;              /* No data blk num yet*/
BH_addr->BQLAH.W = -1;              /* No lda yet         */

BH_addr->BQUSC   = 0;               /* No users yet       */
BH_addr->BQWTC   = 0;               /* No waiters yet     */
BH_addr->BQFLGS  = 0;               /* Clear flag bits    */
BH_addr->BQST    = 0;               /* Clear status bits  */

call ENQH (BFLRU.W, BH_addr);       /* Enqueue to LRU     */
```

5.4 Locking the Buffer LRU (BFLRU.W)

Buffer queue management must ensure that queues are locked when they are manipulated. The buffer LRU is locked for this purpose when Buffer Management sets the generic queue lock bit, bit offset QLOCK, from BFLRU.W. The lock is a spin lock set via a call to BSLOCK.

The generic VSMP locking word for the buffer LRU is LKBLRU.W. LKBLRU.W, as all global, generic locking words, is divided into two separate words. Word 0 is referenced by the lock name itself, and contains the following bit offsets:

```
BPDTRAN (0) - LKBLRU.W transition lock
BPDCHNG (3) - BFLRU.W rescan flag
```

Word 1 has a globally defined index offset, PDUSERS, which in the case of LKBLRU.W, indicates the number of waiters for buffers. It is incremented by the Buffer Management service BLASB (assign system buffer) and by CCB Request Management when a system buffer is requested, but one is not available.

```
XLEF    2,LKBLRU.W      ;LRU lock word.
XNISZ   PDUSERS,2       ;Increment any buffer
NOP                     ;  waiter count.
```

The LKBLRU.W transition lock is acquired for checking the number of buffer waiters and for checking the rescan flag. The rescan flag is set by Buffer Management release system buffer routines (RELB, RELM and RELD), which indicates to assign system buffer routines (ASBUF/BLASB) that a buffer was freed. Below, a portion of RELB illustrates this point.

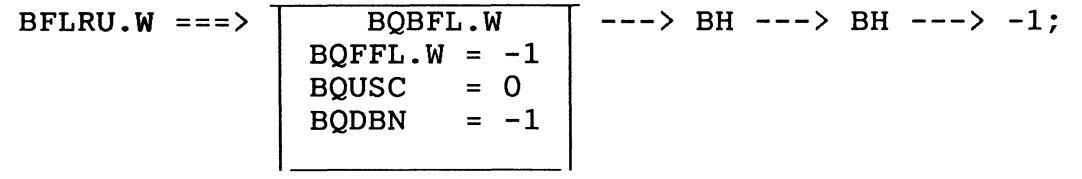
```
RELB:           :
                :

        XLEF    2,LKBLRU.W      ;LRU lock word.
        NLDAI   BPDTRAN,1       ;Transition lock bit.
        XPSHJ   XLOCK           ;Get transition lock.
        NLDAI   BPDCHNG,0       ;Rescan flag (bit).
        WBTO    2,0             ;Set rescan flag.
        XNLDA   0,PDUSERS,2     ;Get buffer waiters.
        WSNE    0,0             ;Are there any?
          WBR   DONE            ;No.

        XJSR    LCBWU           ;Yes, unpend an IOCB.
        WRTN                    ;IOCB unpended.  Return.
                                ;No IOCBs were waiting!
        NLDAI   SKBUF,0         ;Base level pend key.
        XJSR    UNPEND          ;Unpend base level waiters.

DONE:   WBTZ    2,1             ;Release transition lock.
        INTEN                   ;Enable interrupts.
        WRTN                    ;Buffer released!
```

5.5 System Buffer Manipulation

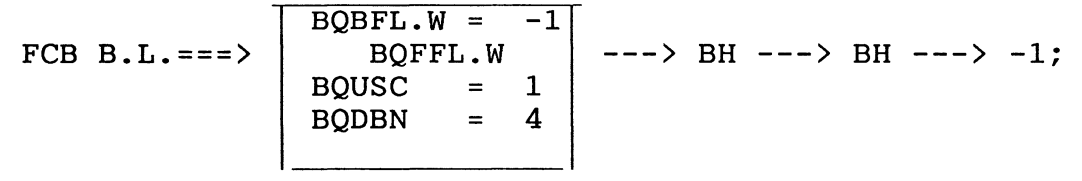
A buffer remains on the buffer LRU as long as its use count is zero. Whenever a system buffer is assigned, its use count is incremented and it is dequeued from the buffer LRU. When CCB Request Management assigns a buffer (for read system buffer command), it is also dequeued from the LRU and its current FCB/LCB buffer list (if it is on one) and then enqueued to the new FCB buffer list. (The LCB cache buffer list forward and backward links are defined by LCB parameters LBCLP.W and LBCLB.W, respectively.) Base level assigners do not enqueue the buffer header to the FCB/LCB buffer list, but enqueue it directly to the disk unit via the Buffer Management services NQBHR/NQBH1. File Management services dequeue buffers from the FCB and enqueue them to the LCB cache buffer list upon the last file close. At release, the buffer is enqueued to the LRU if its use count reaches 0. However, it is not removed from the FCB/LCB buffer list, but must remain there so that the chain may be searched even when the buffer is not in use. This explains why it is possible for the buffer to be enqueued to both the LRU and the FCB/LCB buffer list at the same time. When a file is deleted, all of its buffers on the FCB/LCB buffer lists are invalidated (BQDBN and BQLAH.W set to -1) and enqueued "fresh" to LRU.

The following sequence of events illustrates what happens to a system buffer header used to read in a directory data block.

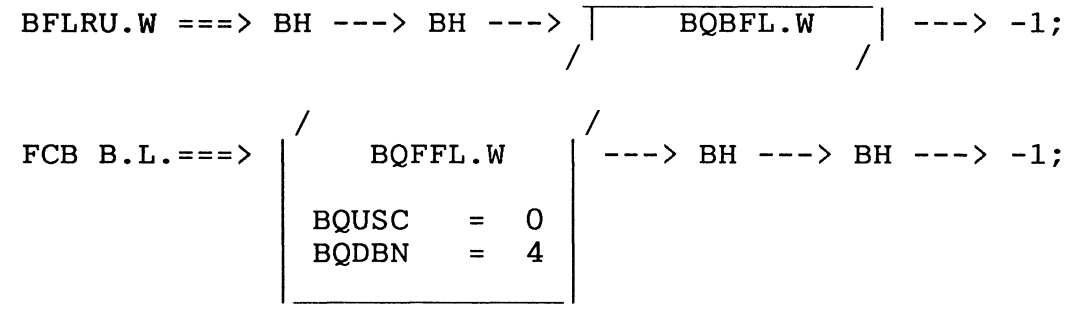
(1) System initialization. Buffer begins on buffer LRU.



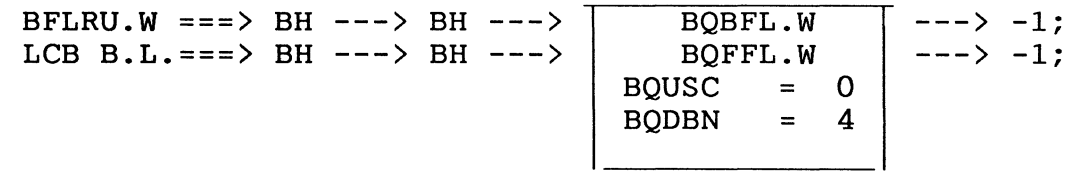
(2) Directory ANYDIR opened. BLKIN called to read in block 4. Buffer dequeued from BFLRU.W and enqueued to FCB buffer list during CCB Request Management processing of read system buffer command.



(3) The buffer is released (RELB), but ANYDIR remains open. The buffer remains on the FCB buffer list, but gets enqueued to the BFLRU.W tail. (Assigned buffers are removed from the head.) If another code path requests ANYDIR block 4, it will be found on the FCB, eliminating CCB Request Management's role of reading it in from disk.

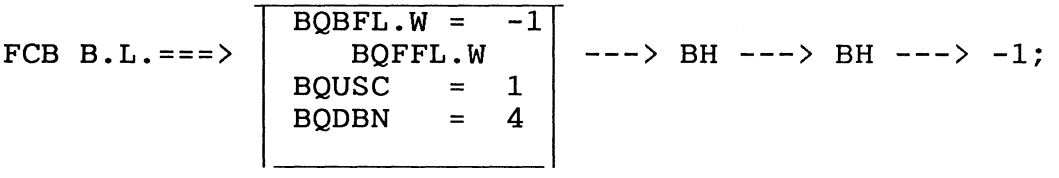


(4) ANYDIR is closed. The buffer is moved from the FCB to the LCB buffer list tail.

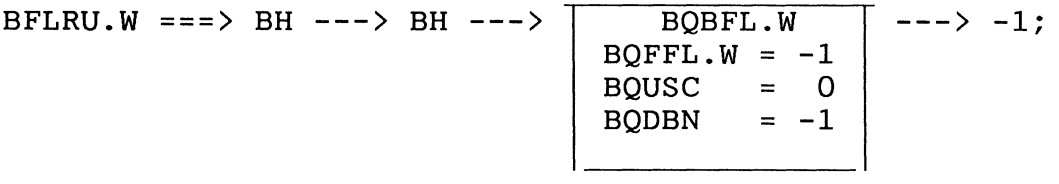


There are now three possibilities that will determine the fate of the buffer:

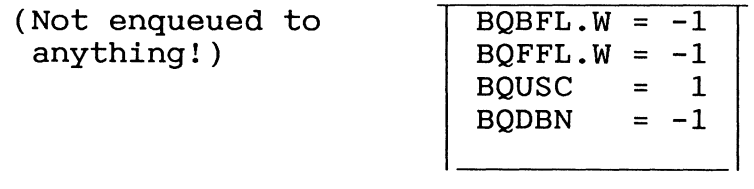
(5a) ANYDIR is reopened and block 4 is requested again (by BLKIN). CCB Request Management will find the buffer on the LCB cache buffer list and move it to the FCB buffer list. The buffer will be dequeued from the LRU and its use count will be incremented. We are now effectively back at step (2) in this procedure.



(5b) ANYDIR is deleted. The buffer header remains on the LRU, but dequeued from the LCB buffer list. This must be done because the data for a non-existent file is useless. The data block number and LDA it represented are invalidated.



(5c) The buffer header reaches the head of the LRU and some code path assigns it. The buffer is removed from all queues. The caller of ASBUF/BLASB (assign BH) is responsible for enqueuing the BH to the FCB buffer list. Only CCB Request Management always does this. Other components do not. When the buffer is released, it is re-enqueued to BFLRU.W. (The illustration below shows the buffer header directly after the call to ASBUF/BLASB.)



When a system buffer is enqueued to the FCB, the FCB address is stored at BH offset BQFCB.W. When it is enqueued to the LCB, the LCB address is stored at the same offset, but it is referenced with the parameter BQLCB.W. Bit 0 is set in BQLCB.W as well, which tells Buffer Management that the buffer header is indeed enqueued to the LCB cache buffer list.

A trivial but extremely important point in the management of system buffer headers cannot be overlooked. Buffer headers can be enqueued to two separate queues simultaneously, implying that two separate queue (double) words must exist for the forward and backward links. When the buffer header is accessed via the buffer LRU, whose forward link is offset 0, the remaining offsets are accessed normally from the offset 0, the beginning address of the buffer header. However, when the buffer header is accessed via the FCB/LCB buffer list, whose forward link is offset 4, the beginning address of the buffer header becomes offset 4. Consequently, the parametric value of each offset following BQFFL.W should actually be 4 less to be referenced correctly from the "new" BH address. Buffer Management implements this by subtracting the value of BQFFL.W (4) from each offset when the BH address is retrieved from the FCB/LCB buffer list (using ECLIPSE MV/Family queue instructions). For example, when BLKIN searches a file's FCB buffer list for a match on the relative data block number, the BH offset loaded into AC3 before the NFSE queue instruction is:

```
        NLDAI   BQDBN-BQFFL.W,3         ;Offset from FCB forward
AGAIN:  NFSE                            ;link to block num word.
          WBR NOBUF                     ;Buffer not found.
          WBR AGAIN                     ;Search interrupted.
          WBR FOUND                     ;Buffer found
```

If the buffer header is found, the "real" buffer header address is calculated and subsequently used:

```
FOUND:  WNADI   BQBFL.W-BQFFL.W,1       ;Add 4 to BH addr
        WMOV    1,2                     ;and move to index reg.
```

The buffer use count is incremented by the occurrence of either of two events: the BH is "assigned" or it is searched for and found on the FCB/LCB buffer list. The use count cannot be incremented if the buffer header is locked or there is I/O in progress. If this is the case, the code path attempting to "gain control" of the buffer must pend until the specific buffer header becomes free. The pend key is the buffer header address (for IOCBs and CBs). The waiter count at offset BQWTC must be incremented as well. When the code path already using the buffer releases it, the waiting code path will be unpended.

A special BH parameter, BQRTC.W, is a doubleword defined over BQUSC and BQWTC. If BQRTC.W is 0, i.e., there are no users and no waiters, this indicates that the buffer header is already enqueued to the buffer LRU. Both ASBUF and BLKIN test BQRTC.W to determine whether or not the system buffer should be dequeued from BFLRU.W before incrementing the use count.

The data block number in the file to which the buffer data corresponds is stored at BH offset BQDBN. This is filled in by CCB Request Management when processing the read system buffer command. The requested block number is input in the CCB and simply transferred to the buffer header. Buffer Management does not actually deal with block numbers, but with logical disk addresses. The reasons for maintaining the data block number in the buffer header are:

1) the FCB buffer list can be searched for specific directory blocks by their relative block number, and

2) the block number is used in creating intra-directory pointers (IDPs).

The logical disk address of the data is stored at BH offset BQLAH.W before Buffer Management is called to enqueue the BH to the appropriate UDB. CCB Request Management traverses a file's index structure in order to fill in this field. Base level callers must get the logical disk address elsewhere. For instance, Process Management obtains the swapfile starting logical address from the Funny FIB, and makes a direct buffer header request (NQLDRQ). LDU Management gets the logical disk addresses of the LDU Name and ACL blocks from the DIB of the first unit. Before the buffer header is enqueued to the disk unit, Buffer Management converts the logical address in the buffer header to a physical address and stores it in the unit's UDB.

System initialization allocates one block of memory for each system buffer and stores its address at offset BQADR.W. This address remains constant for the life of the running system. The number of blocks to transfer, always 1, is stored in BQNBK. There are, however, two other types of buffer headers for which these fields change. The IOCB pseudo-buffer header uses BQADR.W to hold the user address to which the disk-based data will be transferred. BQNBK is a variable depending mostly on the file's elementsize (CCB Request Management does not transfer more than one element of data per buffer header). As explained in Section 5.9, the Buffer Management service NQLDRQ is used to transfer multiple blocks of data for the system, when the logical disk address of the data is previously known. NQLDRQ builds a buffer header on the stack and fills in BQADR.W with the ring 0 address of a reserved area to receive the data, and BQNBK with the number of blocks (512. byte chunks) to transfer.

Buffer headers are enqueued to the UDB request queue (UDB offset UDCRQ.W) of the disk unit on which the data resides through the buffer header driver enqueue link word (BH offset BQQLK.W). Buffer Management calls the driver's device start routine to begin the data transfer. The specific disk driver is responsible for removing buffer headers off the UDB and transmitting the appropriate commands to the controller, which will ultimately cause the data to be transferred from disk to the memory location in BQADR.W.

When the disk driver completes the data transfer, the host is interrupted, and the disk's interrupt service routine is invoked. The interrupt service routine makes a call to the buffer header post-processor, stored at BH offset BQUPD.W. Since the I/O is complete and the code path that summoned Buffer Management to enqueue the buffer header pended itself, the main purpose of the buffer header post-processor is to notify the waiter that the data transfer is finished. This is done by unpending either the waiting CB or IOCB. The post-processor must also unpend CBs or IOCBs, which did not enqueue the I/O, but which are waiting on the particular buffer, perhaps because it was found on the FCB chain with I/O-in-progress. When a flush-buffer request is issued, the requestor enqueues the I/O without pending afterwards. When the I/O completes, the post-processor must decide which object, if any, gets control of the buffer.

There are four separate buffer header post-processing routines. See Section 5.10 for more.

BH Post Processor	Function
PPBSY	General system buffer post-processor.
PPBMD	System buffer post-processor for modified buffers.
PPCPY	Mirror bulk copy buffer header post processor.
RESTR/ RSTRB	IOCB pseudo-buffer header post-processor.

The buffer header status word at offset BQST holds both general status data and the BH request type. The buffer header status bits as defined in PARFS.SR are the following:

```
BSMOD (0) = Buffer modified.  Set by RELM.  Buffer flushed to
            disk when reassigned.  Also signifies "write"
            command.
BSIOP (1) = I/O in progress.  Set by NQBHR/NQBH1.
BSMSI (2) = Modified sector I/O request.
BSERR (3) = 1 if error, otherwise 0.
BSFLS (4) = Flush buffer on last release.  Set by RELF.
BSCDR (5) = Clear disk request.  Tells controller to zero part
            of the disk.
BSNAB (6) = Not a system buffer.  Set if IOCB pseudo buffer
            header or NQLDRQ stack buffer header.
BSPIO (7) = Physical I/O request.
BSLKB (8) = Buffer Header ("long term") lock bit.  Set before BH
            flushed, reset after.  Potential users of this BH
            pend until the lock is freed.
BSEOV (9) = Enable VFU load or override LEOT logic (LPU or MTU).

BSRT1 (11)= Request type bit 0.
BSRT2 (12)= Request type bit 1.
BSRT3 (13)= Request type bit 2.
BSRT4 (14)= Request type bit 3.
BSRT5 (15)= Request type bit 4.
```

Buffer Management references these bit positions in various ways. PARFS.SR contains two other parameter definitions for different types of access. For example, bit position BSLKB (bit 8 in BQST) is usually referenced through another parameter, BQLKB, whose value is the bit offset from the first address of the buffer header. This method is useful in setting and clearing this one bit:

```
        NLDAI   BQLKB,1
        XLEF    2,BH_addr
        WBTZ    2,1
```

The bit position can also be tested by masking it out of the status word. Each bit position has a mask value defined as well, e.g., QTLKB masks the long-term lock bit (000200). Another option used for atomic test and set instructions (WSKBO/WSKBZ/WSZBO) is the bit offset from bit 0 of AC0, which is loaded with the status word. When BQST is loaded (narrow) into AC0, it starts at bit 16. of the accumulator, so BSLKB is defined as bit position BSLKB+16. Due to the multiplicity of bit referencing possibilities, only the bit position within the status word has been defined in this manual.

There are two types of buffer header requests: I/O and non-I/O requests. The I/O request types are represented by the following values in the buffer header status word request type bits:

```
QTIO (0)  = Vanilla (read/write) I/O.
QTWRV (1) = Write verify (read after write).
QTWSW (2) = Write single word verify.
QTRMB (3) = Read with modified bit map.
QTMBC (4) = Mirrored bulk copy.
```

The non-I/O request types are represented by the following values in the buffer header status word request type bits:

```
QTSTP (8) = Stop drive (disk) or unload (mag tape).
QTSTS (9) = Get status.
QTREC (10)= Recalibrate (disk) or rewind (mag tape).
QTDLM (11)= Delete mirror.
QTHFM (12)= Set mirror to half mirror.
QTUSM (13)= Set mirror to unsynchronized.
QTSYM (14)= Set mirror to synchronized.
QTTYP (15)= Get unit type (for unicorn printer only).
```

There are three bits defined in the buffer header flag word at offset BQFLGS:

```
BQTRAN (0) = Buffer header transition lock.
BQWAIT (1) = Base level specific BH waiter bit.
BQIOWT (2) = CCB Request Management (IOCB level) specific
             BH waiter bit.
```

The buffer header transition lock is a short-term lock acquired when quick operations must be done on the buffer header. For example, when it is enqueued or dequeued from the buffer LRU, or the use count must be tested, the transition lock is acquired (via MP general spin lock routines FXLOCK, XLOCK, etc.).

If the buffer header must be accessed on a "long-term" basis, the buffer header long term lock (BSLKB) is acquired. This is done when the buffer is flushed to disk. Since the use count is zero and it may be on the LRU, the long-term lock must be set to prevent the buffer from being removed while the flush is in progress. If a base level code path (not interrupt level or CCB Request Management) finds that the buffer header is locked or has I/O in progress and must wait for that buffer in particular, the caller sets BQWAIT (BH bit offset BBQWAIT) and pends. This is done in routine BWAIT. When CCB Request Management must wait for a buffer header for the same reasons, it sets BQIOWT (BH bit offset BBQIOWT) and pends. Separate bit pointers are defined because the buffer header is assigned to IOCB waiters before base level waiters.

5.6 Emergency Shutdown (ESD) and System Buffers

When AOS/VS is shut down abnormally, the emergency shutdown procedure is executed. ESD is executed by inputting a "START 50" command from SCP-CLI when a system panic occurs. ESD accesses system buffers in three major phases:

1) during "File System Restart" processing,
2) during "Flushing Buffers" processing,
3) during "Open File Processing".

When the message "File System Restart" is printed on the operator console, ESD is currently in progress. During this phase, the buffer LRU chain is validated for consistency. If any of the forward or backward links on BFLRU.W is invalid, ESD forces panic 6016. If the number of queue entries on BFLRU.W exceeds the internal count of buffer headers currently on the queue (BUFCN), ESD forces panic 6017. After validation of BFLRU.W, ESD removes all buffer headers queued to UDBs and enqueues them to the tail of BFLRU.W. By the end of this phase of ESD, all system buffers should be enqueued to the buffer LRU.

When the message "Flushing Buffers" appears on the operator console, ESD begins examining buffer headers on BFLRU.W. If the modified bit is set, ESD flushes the buffer to disk. Disk requests that did not complete do not have the modified bit set, so invalid data will not be flushed.

When the message "Open File Processing" appears on the operator console, ESD begins closing all open files. First, ESD searches through all FCB pages and saves the address of the FCB at the deepest level in the system's directory hierarchy. The level of the file within its LDU's directory hierarchy is found at FCB_addr->FBLVL. If the file is open on a system CCB (i.e., FCB_addr->FBSCB.W != 0), ESD calls upon KCCB.P in File Close Services to close the file. If the file is not open on a system CCB, ESD calls KFCB.P (to first decrement the file open count). Remember that when Resolution Services opens a file on a system CCB, the open count in the FCB is not incremented. If the file level is 0, the file is actually an LDU, and ESD calls IRLSE.P in LDU Management to release it. This procedure is repeated until all files are closed. The last file to be closed is the master root LDU, which marks the completion of ESD.

5.7 Buffer Management Global Variables

Global Location	Significance
BFLRU.W	Buffer LRU queue descriptor (head pointer).
BTAIL.W	Buffer LRU queue descriptor (tail pointer).
LKBLRU.W	Buffer LRU lock word.
BFMIN	Total number of system buffers.
BUFLO	Smallest num buffers on BFLRU.W since boot.
BUFCN	Number of buffers currently on BFLRU.W.
NOBUFS	Number of buffer assign attempts when no buffers have been available on BFLRU.W.
NQDBHRS	Number of BHs currently enqueued to disk.
CACHOF	System buffer access on the LCB chain always allowed, except at ESD. Set to -1 during ESD to disable cache buffer checks.

5.8 Assigning System Buffers (ASBUF/BLASB)

In order for a system buffer header to be enqueued for I/O, the requestor must gain control of it by "assigning" it. This procedure involves removing a free buffer header from the buffer LRU as well as from the FCB/LCB buffer list, invalidating the buffer's contents (simply by clearing out the data block number and logical disk address to which the data corresponds), incrementing the use count, and returning the buffer header to the caller.

There are two Buffer Management services that assign buffer headers: ASBUF, the no-wait version that returns to the caller immediately if no buffers are available; and BLASB, the wait version that pends the calling CB until some code path releases a buffer and it becomes free. CCB Request Management calls ASBUF because of the special mechanism employed for pending IOCBs. CCB Management does indeed pend the IOCB if no buffer is available, but ASBUF must be the assignment interface because BLASB only pends control blocks. LDU Management calls ASBUF when pending would hang the system (master LDU initialization). The following charts display the callers of these routines.

Callers of ASBUF

Caller	Effect
IASBU	CCB Request Management routine to assign system buffers for "read system buffer" requestors and for index blocks.
MLDUI	Read and flush each unit's DIB.
RDBBT.P	Read LDU Bad Block Table.

Callers of BLASB

Caller	Effect
DPMIU	Diskette unit initialization to recal diskette and to detemine its density.
RDNAC.P	Read LDU Name Block and ACL Block.
MIRROR.P	Read the DIB of first unit in mirrored LDU. Read in mirrored LDU Name Block.
READDIB	Read DIB of units in mirrored LDU.
WRITEDIB	Write DIB of units in mirrored LDU.
UPDDIB	Update DIB of units in mirrored LDU.
GETBITS	Read Mirror LDU Bit Map.
GETUDB.P	Delete mirror request if a mirror UDB cannot be allocated.

The system buffer assignment services are straightforward. The C-based algorithm for ASBUF is outlined below. ASBUF returns the buffer header address that the caller now has possession of and is responsible for releasing. If no free buffer is found, ASBUF takes the error return.

```
ASBUF (*BH_addr_out, *found);
{
/*************************************************************
 * Assign a buffer to a base level caller.                   *
 * The LRU must be locked when searching it.                 *
 * The search is for any buffer header without the I/O in    *
 * progress and long-term lock bits set.                     *
 *************************************************************/

   call BSLOCK (&BFLRU.W, QLOCK);

ASBEX:                          /* Label at which search begins */

   call NFSAC (BFLRU.W, BQST, QTIOP+QTLKB, &BH_addr);

/*************************************************************
 * No buffer found.                                          *
 * If the rescan flag was set (by RELB, RELM, RELD), rescan  *
 * BFLRU.W.  If not, record a failure statistic and notify   *
 * the caller that no buffer was assigned.                   *
 *************************************************************/

   if (buffer not found)                /* Search failed.    */
      {
      call XLOCK (&LKBLRU.W, BPDTRAN);  /* Get lock to check */
      if check_bit (&LKBLRU.W, BPDCHNG) /* LRU rescan flag.  */
         {                              /* Rescan flag set!  */
         clear_bit (&LKBLRU.W, BPDTRAN); /* Clear xlock and  */
         goto ASBEX;                    /* rescan BFLRU.W.   */
         }
      else                              /* Rescan flag       */
         {                              /* NOT set.          */
         clear_bit (&LKBLRU.W, BPDTRAN); /* Clear LRU xlock. */
         clear_bit (&BFLRU.W, QLOCK);   /* Clear search lock.*/
          ++NOBUFS += 1;                /* Buff failure stat.*/
         *found = FALSE;                /* Return not found  */
         return;                        /* status to caller. */
         }
      }

/*************************************************************
 * Found a buffer.  Make sure it is free.                    *
 * If there are users or waiters, or the long-term lock is   *
 * held (being flushed), it is not free to deal out.         *
 * So ASBUF restarts the search from the top.                *
 *************************************************************/

   call FXLOCK (BH_addr, BBQTRAN);      /* Get BH trans lock */

   if ((BH_addr->BQRQC.W != 0) ||       /* Waiters + users?  */
       (check_bit(BH_addr->BQST,BSLKB))) /* Or BH locked?    */
      {                                 /* You bet, sorry.   */
      clear_bit (BH_addr, BBQTRAN);     /* Clear trans lock  */
      goto ASBEX;                       /* and rescan LRU.   */
      }
```

```
/*************************************************************
 * Buffer is free, but was it modified?                      *
 * Modified buffers must be flushed to disk before they can  *
 * be assigned. They were not flushed when released since the *
 * data was maintained in memory.  Modified buffers on the   *
 * LRU were released with RELM call.                         *
 *************************************************************/

   if check_bit (BH_addr->BQST, BSMOD)    /* Free buffer mod?  */
      {                                   /* Yes sir.          */
      set_bit (BH_addr, BQLKB);           /* Set flush lock.   */
      clear_bit (BH_addr->BBQTRAN);       /* Clear trans lock. */
      call FLBUF (BH_addr);               /* Down the tubes,   */
      goto ASBEX;                         /* and restart scan. */
      }

/*************************************************************
 * Finally, a free buffer header is available and assigned!  *
 * Bump the use count for the caller, make its status "ready",*
 * and invalidate its old contents.                          *
 *************************************************************/

   BH_addr->BQUSC += 1;                   /* One user so far.  */
   BH_addr->BQST   = 0;                   /* Status = ready.   */
   BH_addr->BQDBN  = -1;                  /* No dbn yet.       */
   BH_addr->BQLAH.W= -1;                  /* No lda yet.       */

/*************************************************************
 * BH in use, so it must be dequeued from the LRU.           *
 * ASBUF dequeues it from either the FCB or LCB buffer list  *
 * as well, since its contents are now invalid.  It is       *
 * possible that the BH is on neither FCB/LCB list, but UNLBH *
 * handles that case by ignoring it.                         *
 *************************************************************/

   call DQBCN (BH_addr);                  /* Dequeue from LRU! */

   if (BH_addr->BQLCB.W & 020000000000)
      call OFFLCB (BH_addr->BQLCB.W, BH_addr); /* DQ from LCB */
   else                                   /*     or            */
      call UNLBH  (BH_addr->BQFCB.W, BH_addr); /* DQ from FCB */


/*************************************************************
 * Buffer assigned.                                          *
 * Clear the BH transition lock and the BFLRU.W search lock. *
 * Return the BH address.                                    *
 *************************************************************/

   clear_bit (BH_addr, BBQTRAN);          /* Clear BH xlock    */
   clear_bit (&BFLRU.W, QLOCK);           /* Clear LRU lock    */

   *BH_addr_out = BH_addr                 /* Return BH addr.   */
   *found = TRUE;
   return;                                /* That's all folks! */

} /* end of ASBUF */
```

5.9 Enqueuing Buffer Headers for Disk I/O (NQBHR/NQBH1)

After a system buffer is assigned, it must be initialized. The caller of ASBUF need only determine the logical disk address (LDA) of the data to be accessed (BQLAH.W) and set the type of request (BQST). The data address (BQADR.W) is calculated at system initialization and remains static. The number of blocks to transfer (BQNBK) for system buffers is always 1. CCB Request Management must compute these values given the CCB input parameters. The Buffer Management services that enqueue a buffer header to a UDB are called NQBHR and NQBH1.

Both NQBHR and NQBH1 enqueue an input buffer header to the appropriate unit in the LDU. The unit to which the request must be sent is determined by the logical disk address (LDA) of the data.

NQBHR determines the unit on which the logical address falls by comparing BH_addr->BQLAH.W with UDB_addr->UDLAH.W on each unit (beginning with the first). If the logical disk address (LDA) specified in the buffer header is less than the last logical disk address on the unit, the data will be found on that unit. The buffer header is enqueued to the UDB and the driver initiates the I/O to the device. Since the LCB points to the first unit in the LDU, its address becomes the input parameter in AC1.

NQBHR is called when the unit on which the logical disk address (LDA) of the request is unknown. There are essentially three circumstances that necessitate this service.

1) Enqueue a general logical disk request
   a. via CCB Request Management, called because the requestor did not know the logical disk address of the data.

   b. via Buffer Management (NQLDRQ service), called because the requestor already had the logical disk address of the data.

2) Flush modified Buffer Headers to disk
   a. from FCB Buffer List on file last close.
   b. from FCB Buffer List on LDU release.
   c. to service a RELF (release and flush) BH request.
   d. to service a RELM (release modified) BH request when a modified buffer is dequeued from the LRU during ASBUF.

3) LDU initialization - Read Name Block and ACL Block.

In all the above cases, the logical disk address of the requested data is already in the buffer header at offset BH_addr->BQLAH.W when the NQBHR call is made. CCB Request Management calculates it during file index level traversal. Any valid buffer header to be flushed must already have the data logical address stored within. LDU Management retrieves the LDU Name and ACL Blocks' logical disk addresses (LDAs) from the first unit's DIB. The only other case which warrants further discussion is the Buffer Management service NQLDRQ.

NQLDRQ, eNQueue a Logical Disk ReQuest, is the Buffer Management service that enqueues a buffer header for external, non-File System components. The NQLDRQ : NQBHR relationship is analogous to the NQCRQ : NQCCB relationship. For reasons of modularity, NQLDRQ exists to provide a clean interface into the File System. It relieves the external caller of the responsibility of explicitly assigning, initializing and enqueuing a buffer header.

There is one additional feature of NQLDRQ that makes its existence essential. While a system buffer allows for a maximum transfer of one disk block, NQLDRQ services buffer header requests for multiple blocks of data. NQLDRQ accomplishes this by NOT enqueuing a system buffer, but by allocating space for a buffer header on the stack. The caller of NQLDRQ sets up a packet with the buffer header request specifications (the only argument). NQLDRQ moves the packet information, which includes the address of a reserved ring 0 memory location and the number of blocks to transfer, to the buffer header on the stack. NQLDRQ invokes NQBHR to enqueue the buffer header to the appropriate UDB and pends (calls BWAIT). Finally, when NQLDRQ is done, it exits normally with a WRTN and the buffer header disappears with the stack.

NQLDRQ is the buffer management equivalent of the File Management NQCRQ service for [system] read and write requests. Its calling routines are listed below. Note that although the physical read/write block system call services are part of the File System, they make use of the NQLDRQ's ability to transfer multiple blocks to a memory address supplied by the caller.

Callers of NQLDRQ

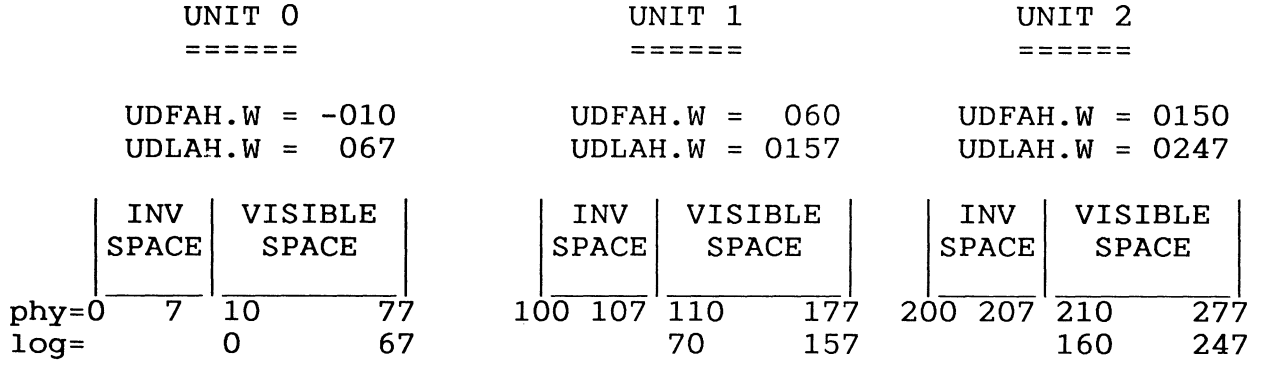
Caller	Reason
PRDB.P	Physical I/O read block system call.
PWRB.P	Physical I/O write block system call.
READSY	Read a system non-resident page from the overlay area on disk.
SWAPIO	Swapfile I/O.
READEXT	Read process table extender from swapfile.
WRITEEXT	Write process table extender to swapfile.

For I/O type requests, NQBH1 assumes the caller already knows the unit on which the logical disk address (LDA) provided in the buffer header falls. Therefore, no search for the proper unit must be done. Instead of the LCB address, the UDB address is input in AC1. The buffer header is enqueued directly to the specified UDB. Non-I/O type requests do not require a logical disk address (LDA), since the command in the status word is sufficient information for the disk controller. NQBH1 is called numerous times only by LDU Management and disk drivers to:

1) Read and write a physical unit's DIB,
2) Service unit recalibration requests (RECAL),
3) Service get unit status requests (GSTS).

Since only the mother processor is able to enqueue disk I/O (see Section 4.2.2), NQBHR/NQBH1 must verify that the caller is running on the mother JP before the buffer header is enqueued to the UDB. If the mother JP is not running, the buffer header is enqueued to the global buffer header wait queue, BHWQ.W, through the BH driver enqueue link word, BH_addr->BQQLK.W. This offset normally represents the BH link when it is enqueued to the UDB for disk I/O, but it is used as the BHWQ.W link as well. Buffer Management then unpends the Disk Manager Task (which ALWAYS runs on the mother JP), who simply issues the NQBHR.

There is one important routine, not expanded in the NQBHR pseudo-code, that deserves further explanation. STUNT, start unit, is called by NQBHR when the request to be enqueued to the UDB is the first on the UDB's BH request list, UDCRQ.W. Since physical units understand only physical disk addresses, this address must be calculated and stored in the UDB (UDDAH.W) along with the number of blocks to transfer (UDNBK). Furthermore, if the request runs off the end of physical unit, STUNT splits the request over multiple units. This is done by putting the first logical disk address (LDA) of the next unit in BQLAH.W and the number of blocks for the second transfer in BQNBK. When the disk interrupt service routine realizes that there are still more blocks to transfer by checking BQNBK (really a special case), it will enqueue the BH to the next unit (if it exists). Following is an illustration of a multiple unit LDU and its logical/physical address relationships:



The physical address of a request is calculated by applying the following formula:

```
UDB_addr->UDDAH.W = BH_addr->BQLAH.W - UDB_addr->UDFAH.W;
```

UDB_addr->UDLAH.W is the unit's last logical disk address (LDA), and UDB_addr->UDFAH.W is the unit's first logical disk address (LDA). Notice that the first logical disk address compensates for invisible space; it is simply 010 (invisible space blocks) less than the first valid logical disk address on the unit. The following examples serve as clarification. A request for logical disk address 0 would translate into physical address 010 (0 - (-010)) on unit 0. A request for logical disk address 070 corresponds to physical address 010 (070 - 060) on unit 1. In order to read/write the DIB in physical block 3, the logical disk address of the request must be specified as -7, since -7 - (-010) = 3. (LDU Management sometimes changes UDFAH.W to 0, so the actual physical address of the DIB can be specified!)

The inputs and outputs of NQBHR/NQBH1, as well as the C-based algorithm for NQBHR, are illustrated below.

Inputs/Outputs for NQBHR and NQBH1

Variable	Input	Output
AC0	BH Post Processor address.	UDB address.
AC1	LCB address (NQBHR). UDB address (NQBH1).	Unchanged.
AC2	BH address.	Unchanged.

```
#define MKDSK 0517   /* Masks out all disk interrupts          */

NQBHR (BH_pp_addr, LCB_addr, BH_addr, &UDB_addr_out)
{
/*************************************************************
 * Set the BH post-processor address.                        *
 * The pp is called from interrupt level to unpend CB or IOCB.*
 *************************************************************/

   BH_addr->BQUPD.W = BH_pp_addr;
```

```
/*************************************************************
 * Enqueue the BH to BHWQ.W if the caller not running on the  *
 * mother JP.  The Disk Manager Task will do the NQBHR!       *
 *************************************************************/

   if (!check_bit (&MYPPCB.W, BCPMST))    /* Mother processor? */
      {                                   /* No. Let mom enque */
      set_bit (BH_addr, BQLKB);           /* Lock BH.          */
      call XLOCK (&IOCBLK.W, BPDTRAN);
      enqueue_to_tail (&BHWQ.W, BH_addr, BH_addr->BQQLK.W);
      clear_bit (&IOCBLK.W, BPDTRAN);
      call DWAKE;                         /* DMTSK will do it  */
      return;
      }

/*************************************************************
 * Set I/O-in-progress and clear error bit.                   *
 * I/O-in-progress should NOT already be set!                 *
 *************************************************************/

   if check_bit (BH_addr, BQTIO)          /* I/O in progress?  */
      call PNIC (14050);                  /* YES?  Fatal error */
   else                                   /* No, so set it ... */
      set_bit (BH_addr, BQTIO);           /* ... now.          */

   clear_bit (BH_addr, BQTER);            /* No errors yet.    */


/*************************************************************
 * Enqueue BH to the UDB on which the requested lda falls.    *
 * Check each UDB on the LCB, starting with the first.        *
 * If the lda is invalid, we will panic!                      */
 *************************************************************/

   for (UDB_addr = LCB_addr->LBUDP.W;
        UDB_addr != 0;
        UDB_addr = UDB_addr->UDUDL.W)
      {
      if (BH_addr->BQLAH.W < UDB_addr->UDLAH.W) /* Is LDA on  */
         {                                      /* this UDB?  */
      /*******************************************************
       * Yes, lda on this UDB.                               *
       * Must mask out all disk interrupts, bump enq BH count *
       * (global and UDB), and update disk metering stats if  *
       * metering is to be done (please see CODE for more).   *
       *******************************************************/

         call MASK (MKDSK);               /* during I/O enque  */
         NQDBHRS += 1;                    /* Global BH enq cnt */
         UDB_addr->UDCQL += 1;            /* UDB    BH enq cnt */
         do metering statistics;          /* Meter meter meter */
```

```
/*********************************************************
 * If this is the first BH on the UDB request list,      *
 * make it the first request and start the phys unit.    *
 * STUNT and the start-up DCT routines start the         *
 * unit/controller and get the request underway.  The    *
 * next time the host hears from this buffer header is   *
 * when the disk interrupts to signal I/O completion.    *
 * (STUNT explained above in this section.)              *
 *********************************************************/

   if (UDB_addr->UDCQL == 1)            /* Only request */
      {
      UDB_addr->UDCRQ.W = BH_addr;

      call STUNT;
      call (UDB_addr->DCT_addr->DCSUP.W);
      call (UDB_addr->DCT_addr->DCSTR.W);
      }

/*********************************************************
 * Not first request to UDB.                             *
 * Let the unit's enqueue routine do the work, since     *
 * enqueue algorithms are disk-specific.                 *
 * On non-unicorn type disks (e.g., DPF), the unit/      *
 * controller are already started, and the BH pp will    *
 * adjust UDDAH.W and UDCRQ.W to start the next BH req   *
 * on its way.  Unicorn types (e.g., DPJ) must be know   *
 * about ALL new BHs when they are enqueued (NOW!).      *
 *********************************************************/

   else
      {
      call (UDB_addr->DCT_addr->DCENQ.W);

      if check_bit (BH_addr, BDUCN)     /* Unicorn dev? */
         {                                        /* Yes */
         call (UDB_addr->DCT_addr->DCSUP.W);
         call (UDB_addr->DCT_addr->DCSTR.W);
         }
      }


/*********************************************************
 * BH enqueued and on its way.                           *
 * "UNMASK" restores old interrupt mask to CMSK.         *
 * Send back the UDB address to caller, and we're done!  *
 *********************************************************/

   call UNMASK;

   *UDB_addr_out = UDB_addr;
   return;

  } /* end of big if */
} /* end of for */
```

```
/*************************************************************
 * If we exit the "for" loop, the logical disk addr requested *
 * was greater than the last logical address on the LDU.      *
 * That is intolerable, so panic.                             *
 *************************************************************/

   call PNIC (14047);

} /* end of NQBHR */
```

5.10 Pending on Buffer Header I/O Completion (BWAIT)

After the buffer header has been enqueued to the UDB by NQBHR, the caller must await I/O completion. Buffer Management provides a service to pend the waiting control block after issuing NQBHR, NQBH1 or NQLDRQ. This service is called BWAIT.

Since it is possible for the I/O to complete even before the control block calls BWAIT, the I/O-in-progress bit and the BH lock bit must be checked before the actual pend. If either bit is set (I/O incomplete), BWAIT sets the waiting on this buffer header bit (BBQWAIT) and calls the Process Management service MPEND, which removes the CB from ELQUE and places it on PELEMQ.W.

When I/O finally completes, the buffer header post processor will unpend the CB, and BWAIT will pass control back to its caller. The requested disk data is now accessible in memory through the buffer header data address BH_addr->BQADR.W.

5.11 Releasing System Buffer Headers

After the caller of ASBUF/BLASB possesses the returned buffer header address, the buffer is removed from the LRU and its use count is incremented. It is the responsibility of the assigner to undo these actions so the buffer will become free for general system use. There are four Buffer Management services to "release" (or free) system buffer headers.

Variant	Function
RELB	Vanilla release system buffer.
RELM	Release system buffer and set modified bit.
RELF	Release system buffer and flush it.
RELD	Release system buffer and destroy contents.

After I/O completes on a system buffer and the data address contains the requested disk block contents, the buffer user is free to modify the buffer's contents. If the buffer user decides not to modify the contents, the buffer is released normally (RELB). Directory Management typically does this after reading DDEs.

If the buffer user does modify the contents, the buffer can either be released modified (RELM) or flushed immediately (RELF). Directory bit map blocks are released modified. The blocks will be found on the FCB buffer list and the modified contents will remain in effect. The block is flushed before an assigner attempts to use it and when the file (directory) is closed. If the system crashes before the block is flushed to disk, the FIXUP utility will correctly rebuild the directory. Buffers are released to be flushed when the buffer contains data that must be written to disk immediately. For instance, when a file is created, the buffers containing the new FNB and FIB are released flushed.

Buffers are "destroyed" (RELD) when their contents are no longer valid. The most common case occurs during file delete and truncate processing. The system buffers that hold deleted index blocks are released and destroyed, since those index blocks will never (and should never) be referenced again. LDU Management destroys the buffer containing the DIB of a released LDU. The driver of DPM type diskettes assigns a system buffer just to make a quick recal request and read a block of data. When it's finished, it does a RELD. Upon errors from NQBHR/NQBH1, buffers are often released and destroyed. The buffer's content is invalidated by filling the data block number (BQDBN) and logical disk address (BQLAH.W) fields with -1.

The C-based algorithm for all Buffer Management buffer release services is illustrated below. Since most of the code is common, a case statement allows special pre-processing to be done for the specific version called. The only input is the buffer header address.

```
RELB (BH_addr)                          /* In: BH address    */
RELD (BH_addr)                          /* In: BH address    */
RELF (BH_addr)                          /* In: BH address    */
RELM (BH_addr)                          /* In: BH address    */
{
/*****************************************************************
 * RELEASE A SYSTEM BUFFER.                                      *
 *****************************************************************/

   switch (release routine)
      {
      case RELF:                        /* RELEASE AND FLUSH */
         set_bit (BH_addr, BQTFL);      /* Set BH flush bit  */
         set_bit (BH_addr, BQTMD);      /* and BH modified   */
         break;

      case RELM:                        /* RELEASE MODIFIED  */
         set_bit (BH_addr, BQTMD);      /* Set BH modified   */
         break;

      case RELD:                        /* RELEASE & DESTROY */
         call XLOCK (BH_addr, BBQTRAN); /* Get BH trans lock */

         BH_addr->BQUSC -= 1;           /* Decr use count.   */
         if (BH_addr->BQUSC != 0)       /* Use count better  *
            call PNIC (14065);          /* be 0 on RELD!!!!! */

         if (BH_addr->BQWTC.W != 0)     /* There better be 0 */
            call PNIC (14112);          /* waiters on RELD!! */

         ENQH (&BFLRU.W, BH_addr);      /* Enque to LRU head */

         BH_addr->BQDBN   = -1;         /* Invalidate file   */
         BH_addr->BQLAH.W = -1;         /* blk num and lda.  */
         BH_addr->BQST    = 0;          /* Clear status word */
         BH_addr->BQFLGS  = 0;          /* Flags (trans lock)*/
         break;

      case RELB:                        /* RELEASE BUFFER    */
         break;                         /* Fall through ...  */
   }

/*****************************************************************
 * Release common code (except for RELD).                        *
 * If the buffer contents are not being invalidated (RELD),      *
 * decrement use count.  If there are no waiters on this BH,     *
 * it can be enqueued to the LRU TAIL to give it most time to    *
 * be found on FCB/LCB.  RELD was enqueued to the HEAD so it     *
 * will be the first one assigned, since its contents are no     *
 * longer valid.                                                 *
 *****************************************************************/
```

```
   if (release routine != RELD)
      {
      call FXLOCK (BH_addr, BBQTRAN);     /* Get BH trans lock */

      if (--BH_addr->BQUSC == 0)                /* Use count 0? */
         {                                      /* Yes.         */
         if (BH_addr->BQWTC == 0)               /* Any waiters? */
            call PENQT (&BFLRU.W, BH_addr);     /* No - to LRU. */
         }

/*************************************************************
 * If this is a RELF, BQTFL will be set, and the buffer will  *
 * be flushed.  If the caller left the BH locked, the buffer  *
 * will be flushed.  The bits will be turned off when the     *
 * flush completes by the BH post processor PPBMD.            *
 *************************************************************/

      if (check_bit (BH_addr, BQTFL) ||         /* Is buffer to */
          check_bit (BH_addr, BQLKB))           /* be flushed?  */
         {                                      /* Yes.         */
         clear_bit (BH_addr, BBQTRAN);          /* Clear xlock. */
         call FLBUF (BH_addr);                  /* Flush it.    */
         return;                                /* All done!    */
         }
      }

/*************************************************************
 * Common code.                                               *
 * Clear BH transition lock and get BFLRU.W transition lock.  *
 * If CCB Management was waiting for a BH to become free,     *
 * wake up one waiting IOCB before checking base level.       *
 * If no IOCBs waiting, wake up all base level waiters, who   *
 * will have to fight it out in ASBUF to get it.              *
 *************************************************************/

   clear_bit (BH_addr, BBQTRAN);          /* Clear BH trans lock */
   call XLOCK (&LKBLRU.W, BPDTRAN);       /* Get BFLRU trans lck */

   if (LKBLRU.W->PDUSERS != 0)            /* Waiters for any BH? */
      {                                   /* Yes.                */
      call LCBWU (&buffer_assigned);      /* Try waking an IOCB. */

      if (! buffer_assigned)              /* Was one waiting?    */
         call UNPEND (SKBUF);             /* No, unpend CBs.     */
      }

/*************************************************************
 * Done!                                                      *
 *************************************************************/

   clear_bit (&LKBLRU.W, BPDTRAN);        /* Release trans lock  */
   return;                                /* and back to caller. */

} /* end of REL* */
```

5.12 System Buffer Header Post-Processing

Buffer header post-processing is called from (disk) interrupt level by subroutine IODON. NQBHR/NQBH1 stored the address of the appropriate BH post-processor in the BH when the request was enqueued. The four BH post-processors were outlined in Section 5.5.5. As with the CCB post-processors, the main objective of the BH post-processors is to unpend the CB or IOCB awaiting request completion. Some buffer header management is done as well. Since the most common routine is PPBSY, the generic post-processor called when a general buffer header request is made (from base level), it is illustrated algorithmically below.

```
PPBSY (BH_addr)
{

/*************************************************************
 * Get transition lock while modifying BH status.            *
 * I/O is complete, so clear the modified, I/O in progress,  *
 * flush and BH long-term lock bits.  Leave the other status *
 * bit alone, though.                                        *
 *************************************************************/

   call XLOCK (BH_addr, BBQTRAN);
   BH_addr->BQST &= -1-QTMOD-QTIOP-QTFLS-QTLKB;

/*************************************************************
 * Now check if any CBs were waiting for this BH.            *
 * BWAIT, called after NQBHR for CBs to pend, sets BBQWAIT   *
 * and uses the BH address as the pend key.  If the bit is   *
 * set, clear it and wake up the CB.                         *
 *************************************************************/

   if check_bit (BH_addr, BBQWAIT)       /* CB waiting on BH?*/
      {                                  /* Yes.             */
      clear_bit (BH_addr, BBQWAIT);      /* Clear waiting bit*/
      call UNPEND (BH_addr);             /* And unpend CB.   */
      }

   clear_bit (BH_addr, BBQTRAN);         /* Clear trans lock */
   return;                               /* and leave.       */

} /* end of PPBSY */
```

5.13 Physical Disk User Read/Write Services

A physical disk opened as such (not part of an LDU) is treated as an enormous file of user accessible contiguous data. No AOS/VS file structure is assumed to exist. The concept of visible and invisible space is also invalid. As far as the operating system is concerned, logical disk addressing is not understood.

The system call services that initiate physical I/O are PRDB.P (read) and PWRB.P (write). When physical I/O is performed, the start address of the requested file is actually the first physical address of the disk unit, block 0. Since the desired physical address is automatically available, a direct buffer header request can be made. PRDB.P/PWRB.P use the Buffer Management NQLDRQ service instead of NQBH1, because the FCB of the disk unit contains only the LCB address, not the UDB address. (In the case of a disk opened as a separate physical unit, an LCB is allocated anyway, and the only unit's UDB is enqueued to it.)

Below is the C-based algorithm for PRDB.W/PWRB.W. The user's input is the channel number in AC1 and the packet address in AC2. Much of the code is common with the logical disk I/O system call services RDB.P/WRB.P (Section 3.4.5).

```
#define SCDCG 0144

PRDB.P/PWRB.P (TAC1.W, TAC2.W)
{
   /*********************************************************
    * This is a system call with a packet.                  *
    * Get the necessary data into system space.             *
    *********************************************************/

   channel_num = TAC1.W;
   caller_pkt  = TAC2.W
   wblm (&caller_pkt, &sys_pkt, pkt_len);

   /*********************************************************
    * Find the (user) CCB for the input channel.            *
    * DFAULT.P returns error code n CERWD of control block, *
    * which the system call processor will understand.      *
    *********************************************************/

   PTBL_addr = *CC.W->CPTAD.W;
   call DFAULT.P (channel_num, PTBL_addr, &CCB_addr);
      if (error) return;              /* Error code in CERWD */
```

```
/************************************************************
 * Got the CCB.  Do validity checking.                      *
 * All error returns now must unpin the CCB page (which     *
 * DFAULT.P pinned!) and return 0 bytes transferred before  *
 * returning errors.  The first error return does this, the *
 * rest are omitted to save space, BUT THE SAME IS DONE!    *
 ************************************************************/

/* Store num blks to xfer.  Error out on null request.      */
if ((CCB_addr->CBNBK = sys_pkt->PSTI & 0377) == 0)
   {
   call UNPIN (CCB_addr, PTBL_addr);
   TAC1.W = 0;
   return (Invalid_system_call_parameter);
   }

/* Must be a valid block count.                             */
if (sys_pkt->PRNH.W & 037740000000 != 0)
   return (Invalid_system_call_parameter);

/* CCB must not be locked already.                          */
if (get_bit (CCB_addr, BCBLK))
   return (Simultaneous_requests_on_same_channel);

/* Cannot be peripheral type CCB (?FGFN, IPC file)          */
 if (CCB_addr->CBSTS & BCBPE))
   return (Wrong_I/O_type_for_OPEN_type);

/* CCB/FCB unique IDs must match.  Panic 6034 if not!!!     */
call VALCID (CCB_addr);


/************************************************************
 * CCB/Request A-OK so far.                                 *
 * Init the CCB some more.                                  *
 ************************************************************/
TCB_addr = *CC.W->CATCB.W                /* Save TCB addr   */
CCB_addr->CBTCB.W = TCB_addr;            /* TCB addr to CCB */
CCB_addr->CBPTA.W = PTBL_addr;           /* and PTBL addr   */
CCB_addr->CBUAD.W = sys_pkt->PCAD.W      /* and buffer addr */

if (TCB_addr->SYSWD == ?RDB)             /* If ?RDB, CCB    */
   CCB_addr->CBFLG = CBRED;              /* read command;   */
else                                     /* if ?WRB, CCB    */
   CCB_addr->CBFLG = CBWRI;              /* write command.  */

/************************************************************
 * If this is a unit CCB, let the correct unit I/O handler  *
 * finish it.  Long Dispatch to the routine.  The unit type *
 * is stored in bits 11-15 of the CCB status word.          *
 ************************************************************/

if (CCB_addr->CBSTS & BCBUN)
   {
   dispatch_to_unit_handler (CCB_addr->CBSTS & CUMSK);
   }
```

```
   /*************************************************************
    * Check requestor's access to the file.                     *
    * User opens of dir files were given NO access privileges   *
    * when the file was opened, as only AOS/VS, specifically,   *
    * resolution services, can perform directory file I/O.      *
    *************************************************************/

   if (TCB_addr->SYSWD = ?RDB)                /* ?RDB ?          */
      if (! CCB_addr->CBSTS & read_access) /* Yes, read accs? */
         {                                    /* Nope ...        */
         if (CCB_addr->CBFCB.W->FBTYP = DIRECTORY_TYPE)
            return (Illegal_file_type);
         else
            return (Read_access_denied);
         }

   if (TCB_addr->SYSWD = ?WRB)                /* ?WDB ?          */
      if (! CCB_addr->CBSTS & write_access)/* Yes, writ accs? */
         {                                    /* Nope ...        */
         if (CCB_addr->CBFCB.W->FBTYP = DIRECTORY_TYPE)
            return (Illegal_file_type);
         else
            return (Write_access_denied);
         }

/*************************************************************
 * Physical I/O only allowed on physical disk units!         *
 * Calculate that the request fits on the disk unit.         *
 * Since the CCB page was pinned by DFAULT.P, it must be     *
 * unpinned before the error return (steps omitted here).    *
 *************************************************************/

   if (CCB_addr->CBFCB.W->FBTYP != ?FDKU)       /* Phys disk? */
      return (Wrong_I/O_type_for_OPEN_type);    /* No, bad.   */

   final_blk_num = sys_pkt->PRNH.W + CCB_addr->CBNBK;
   if (final_blk_num > FCB_addr->FBDFH.W)       /* Req fits?  */
      return (End_of_file);                     /* No, bad.   */

/*************************************************************
 * Since physical I/O calls NQLDRQ, a Buffer Management      *
 * service, the modified bit must be set here.  This step is *
 * not done in RDB.P/WRB.P because I/O Management sets it    *
 * when processing the CCB request.                          *
 *************************************************************/

   if (TCB_addr->SYSWD = ?PWRB)
      set_bit (FCB_addr, BFBMD);
```

```
/*****************************************************************
 * MFLTPIN: check write access enabled on user pages of PIO      *
 *          status block, and faults and pins them.              *
 * DMTST: validates, faults and pins user buffer pages.          *
 * CHRG: charges caller for I/O and makes time slice checks.     *
 *****************************************************************/

   call MFLTPIN (BIT0 + PIBLT, caller_pkt->PRBB, PTBL_addr);
   call DMTST (CCB_addr->NBLK * 512., CCB_addr);
   call CHRG (SCDCG, CCB_addr->CBNBK, CCB_addr);

/*****************************************************************
 * Enqueue Logical Disk Request: NQLDRQ.                         *
 * The only argument is a packet, built here.  NQLDRQ will       *
 * enqueue a buffer header to the unit's UDB to initiate I/O.    *
 * See preceding description of PIO for packet specifics.        *
 *****************************************************************/

   set_up_packet (nqldrq_packet);          /* Ready packet    */
   call NQLDRQ (nqldrq_packet);            /* and start I/O.  */

   if (error)                              /* Set TCB err bit */
      TCB_addr->SYSWD |= PERB;             /* if err occurred */


/*****************************************************************
 * Data back to user.  Release pinned pages.                     *
 * Move PIO status blk to user packet.  The buffer header PP     *
 * moved the status data to PIO_status_blk, a local address.     *
 * Then, unpin the pages of the user status block, the user      *
 * buffer (pinned by DMTST) and the CCB (pinned by DFAULT.P).    *
 *****************************************************************/

   wblm (&PIO_status_blk, &caller_pkt->PRBB, PIBLT);

   call MUNPIN (PIBLK, caller_pkt->PRBB, PTBL_addr);
   call UNPIN  (caller_pkt->PCAD.W, PTBL_addr);
   call UNPIN  (CCB_addr, PTBL_addr);

   if (TCB_addr->SYSWD | PERB)             /* Error during PIO? */
      error_return;                        /* Yes, code in TCB. */
   else                                    /* No,               */
      return;                              /* Normal return.    */

} /* end of PRDB.W/PWRB.W */
```

6 Logical Disk Unit (LDU) Management

6.1 Overview

A logical disk is an association of physical disks that creates the abstraction of a single, contiguously addressable unit. This extended disk space support is maintained within the operating system by the LDU Management layer of the File System. The functions of LDU Management, each of which will be discussed in detail in this section, can be summarized as follows:

1) To provide a contiguous array of logical disk blocks, which span one or more physical disks and which hide disk blocks used for LDU Management (invisible space);

2) To provide bad block remapping for logical disk blocks, which preserves the contiguous array of logical blocks in spite of isolated bad sectors;

3) To maintain the allocation status of each logical disk block and the status of the LDU as a whole.

The concept of logical disk addressing is fundamental to LDU Management. Logical disk addresses, beginning with 0 and ending with n, correspond to the sequential blocks in the visible space portion of an LDU. Since the invisible space of each PU in an LDU consists of 8. reserved blocks, logical address 0 corresponds to physical address 8. The following diagram illustrates the logical/physical disk address in an LDU consisting of two physical units.

Logical Disk Unit

	Unit 1		Unit 2	
	INVISIBLE	VISIBLE	INVISIBLE	VISIBLE
Phys Addr:	0 7	8 n	n+1 n+8	n+9 2n
Log Addr:	-8	0 n-8	n-15.	n-7 2(n-8)

6.2 Logical Unit Control Block (LCB) Parameter Definitions

Offset		Logical Unit Control Block (LCB)
LBLBP.W	0	LCB list pointer.
LBUDP.W	2	UDB Pointer (to first UDB in LDU).
LBCLP.W	4	LCB Cache Buffer List Queue Descriptor (head).
LBCLB.W	6	LCB Cache Buffer List Queue Descriptor (tail).
LBBML	10	LDU Bit Map Lock.
LBWDL	11	LDU Bit Map Withdraw Lock.
LBSTS	12	LCB status word.
LBTM1	13	Temporary (not used).
LBMFC.W	14	LDU Bit Map FCB address.
LBMBF.W	16	LDU Bit Map FCB Buffer Header address.
LBRCB.W	20	LDU Root CCB address.
LBCSH.W	22	Current space on LDU.
LBMSH.W	24	Maximum space on LDU.
LBBLT	26	Length of LCB.

There is one LCB per logical disk unit. It holds information relating to the LDU it represents. The LCB is allocated from main system memory (GSMEM) and initialized by LDU Initialization. Unit Management allocates an LCB when opening a physical unit as well. Even though physical disk units opened for physical I/O are NOT logical disk units, the LCB performs the function of holding the necessary, single UDB of the unit. Each LCB is enqueued to the tail of the global LCB chain, whose head and pointers are found at AOS/VS global locations .MLCB and .ELCB, respectively. The LCB is linked through offset LCB_addr->LBLBP.W.

One Unit Definition Block (UDB) is allocated for each physical unit in the LDU. The UDBs are linked in unit sequential order through offset LCB_addr->LBUDP.W. LDU Initialization sets up this singly-linked list. When a logical disk request is made, Buffer Management scans the list of UDBs to determine where to enqueue the buffer header. The UDB is illustrated and explained in detail in Section 6.3.

The LCB cache buffer list contains the system buffer headers that had previously been on a file's FCB buffer list. They were put on the LCB cache buffer list when the file was closed. (See Buffer Management for more details.)

The LCB contains two LDU Bit Map lock words. Only CCB Request Management withdraws and deposits disk blocks; therefore, only CCB Request Management can access the LDU Bit Map. LCB_addr->LBBML is the LCB Bit Map global lock. The LDU withdraw and deposit operations must first acquire this exclusive lock before withdrawing or depositing disk blocks in the LDU Bit Map. This means that either a withdraw or a deposit operation may be active at any one time, but not both at the same time. However, even if this lock is free, LDU withdraw operations may not acquire it if the "withdraw lock" is held.

This field can contain the following legal values:

0 = the lock is free
1 = blocks are being withdrawn
-1 = blocks are being deposited

If a withdraw or deposit operation attempts to acquire this lock but finds that it is already held, the currently running IOCB will be pended with pend key IOBMW. When the lock is freed, IOCBs waiting on the lock are unpended. The LDU Management routines that acquire the LDU Bit Map global lock are MWAIT (for withdraw requests) and MSWAT (for deposit requests). This lock is released by the CCB Request Management routine IOWU.

LCB_addr->LBWDL is the "withdraw lock." CCB Request Management Command processing for ?TRUNCATE (CCB commands CBTRN1 and CBTRN2) acquires this exclusive lock to prohibit any withdraws from the Bit Map (i.e., to disallow any LDU disk block allocations). This is because file truncation involves modifying index blocks whose modified contents must be flushed to disk before they can be withdrawn. Consider the following example: A request to truncate a file at data element 65. is enqueued. CCB Request Management deposits all file blocks past data element 65. and zeroes out their corresponding pointers in the system buffer holding the index block. The disk has not reflected the index block modification yet. If, by chance, the system crashed before the index block buffer was flushed to disk, and those previously deposited blocks were since withdrawn (re-allocated), there would be two pointers to the blocks: one in the index block and one "somewhere" else. The withdraw lock eliminates this condition from arising by prohibiting withdraws during file truncation. Withdraws are not prohibited during delete processing because the file's first address (in the FIB) is zeroed out, thus cutting off all ties between the index blocks and the file's contents.

Two conditions must hold true in order for the Bit Map withdraw lock to be set:

1) the Bit Map withdraw lock must be free, and

2) the Bit Map global lock must not be equal to 1, i.e., a withdraw must not be in progress.

If either of the above conditions are true, the currently running IOCB will pend with the pend key IOWWT. When the Bit Map withdraw lock and the Bit Map global lock are freed, IOCB lock waiters on the respective locks are unpended. The LDU Management routines that acquire and release the LDU withdraw lock are MWLCK and MWULK, respectively.

The Bit Map FCB address is stored at LCB_addr->LBMFC.W by LDU Initialization. When LDU Management reads bit map blocks into system buffers, the buffer header address is temporarily saved at LCB_addr->LBMFB.W. These buffers are released modified and flushed back to the bit map when the LDU released.

The LCB status word is found at LCB_addr->LBSTS. The following bit positions are defined:

```
LBFIX  (0) = Must run FIXUP before re-initializing the LDU.
LBFUR  (4) = FIXUP recommended on this LDU.
LBTRAN (5) = LCB transition lock.
```

LBFIX and LBFUR are set dynamically by File Management (in routine FFCB) if an invalid FCB address is found in the FIB of any file on the LDU. This indicates to LDU Release that FIXUP must be run on the LDU and, if the system is releasing the disk because it is shutting down, that an abnormal system shutdown has occurred. (Refer to LDU Release for details.) The LCB transition lock is acquired by Buffer Management operations that search and modify the LCB cache buffer list.

The LCB holds storage for the LDU root CCB address at LCB_addr->LBRCB.W. Space specifications are kept in the LCB as well. The current space available on the LDU is maintained at LCB_addr->LBCSH.W. The maximum space available, static and set by LDU initialization, is found at LCB_addr->LBMSH.W

6.3 Unit Definition Block (UDB) Parameter Definitions

Offset		Unit Definition Block (UDB)
UDDCT.W	0	DCT address of this device.
UDUNT	2	Device unit number.
UDCRQ.W	3	Unit Request list. Chain of enqueued BHs.
UDLAH.W	5	Unit Last Logical Address.
UDUDL.W	7	UDB Forward Logical Link.
UDUDP.W	11	UDB Forward Physical Link.
UDFAH.W	13	Unit Start Address.
UDNBK	15	Number of blocks to transfer for next request.
UDSTS	16	UDB status word.
UDCYS	17	Cylinder size in sectors.
UDNSC	20	Number of sectors per track.
UDNHD	21	Number of disk heads.
UDDAH.W	22	Physical Disk Address of request data.
UDDOA	24	Driver-specific DOA word.
UDDOC	25	Driver-specific DOC word.
UDERC	26	Error Counter.
UDERF	27	Error Flags.
UDUNS	30	Driver-specific DIA/DIC words.
UDSTB	31	Driver-specific DIB word.
UDFLG	32	UDB Flag word.
UDETY	33	Another error flag.
UDBBT.W	34	Bad Block Table address.
UDRMH.W	36	Physical disk address of data in Remap Area.
UDINT	40	Controller inference flag.
UDCQL	41	Current Buffer Header Queue length.
UDMET.W	42	Metering area address.
UDEST	44	Device error logging words ...
UDRTY	45	Error status (bits 0-7), retry count (bits 8-15).
UDTBK	46	Number of blocks to transfer for this request.
		:
UDEOA	74	DOC word on fatal error.
UDEOC	75	Cylinder number of fatal error.
UDDIA	76	DIA/DIC word on fatal error.
UDDTB	77	DIB word on fatal error.
UDBLT	100	Length of UDB.

One UDB is allocated from main memory (GSMEM) by Unit Management for each open physical unit in the system. The UDB contains unit specific data. It points to the device's Device Control Table (DCT), which holds essential device-specific data. The UDB is accessed and modified primarily by specific device drivers. UDBs are allocated for non-disk units, such as magnetic tapes, multiprocessor communications adaptors, and line printers. For this reason, the parameter descriptions in the diagram above will not be explained here; they have been and will be presented separately by the operating system component that uses the parameters.

When a physical unit is specified as part of an LDU, and the LDU becomes initialized, the UDB is linked both to the DCT physical unit list (DCT_addr->DCPUL.W) through UDB_addr->UDUDP.W, and to the LCB (LDB_addr->LBUDP.W) through UDB_addr->UDUDL.W. When a physical unit is opened separately for physical I/O, the same procedure is followed, despite the fact that the unit is not part of an LDU. This mechanism allows for physical I/O service routines to use the same Buffer Management services that enqueue disk I/O. AOS/VS determines that any physical unit is opened ("Device in use") by searching for a UDB that describes the unit on the DCT physical unit list.

6.4 LDU Initialization

A logical disk is created by the DFMTR utility. Physical and logical disk unit information that ties the units together is stored in physical block 3 (the DIB) of each unit. A newly created LDU contains no directory sub-structure; it consists solely of itself, the LDU root directory. The result of LDU initialization is the appearance of the LDU name in the existing AOS/VS directory hierarchy, and the ability for system users to "dir into" the LDU and perform (valid) file operations within it.

LDU initialization is essentially equivalent to creating directory data entries in the LDU's parent directory and then "opening" the LDU. However, since one logical disk is not necessarily limited by the boundaries of one physical disk, each physical unit in the LDU must be "opened." File Open Services does not provide a variant for LDU opens, but LDU Management explicitly calls File Management routines to create the databases necessary to initialize the LDU. The following databases are allocated:

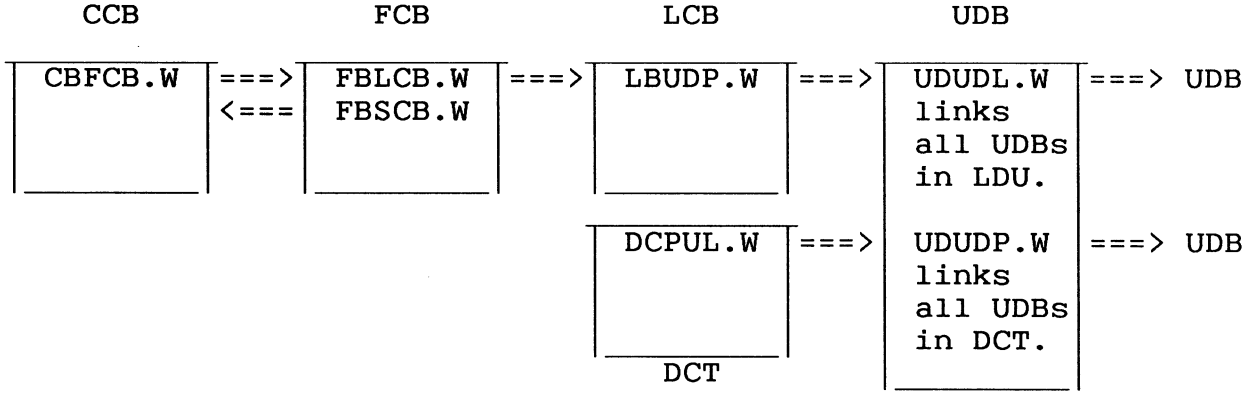
1) One FNB, FIB, and FAC for the LDU
2) One FCB for the LDU
3) One system CCB for the LDU
4) One LCB for the LDU
5) One UDB for each unit in the LDU

6) One MDB, if the LDU is mirrored
7) One MURB for each unit, if the LDU is mirrored
8) One mirror UDB for each unit, if the LDU is mirrored

The DIB of the first unit of the LDU contains the addresses of the LDU name and ACL blocks. In order to make the LDU visible to the user community, its name must be "grafted" onto a specified directory (an option in the ?XINIT packet). This operation is accomplished by creating an FNB entry in the directory. Even though the LDU's in-core Funny FIB (in the FCB) is flushed to the DIB upon release, a FIB entry must be created in the directory to hold the universal ACL and to point to a FAC. An LDU can be initialized by any user possessing write or append access to the directory into which the LDU is to be grafted and owner access to the LDU's root directory.

LDU Management calls GFCB to allocate an FCB and GSMEM to allocate a system CCB for the LDU. The FCB contains the Funny FIB, which was read in from the LDU's DIB. Since the initialized LDU is accessible to the entire user community, not exclusively to the process that issues the ?XINIT, a system CCB must be allocated. LDU Management directly calls on the Memory Management service GSMEM to allocate the memory. The system (root) CCB is stored at LCB_addr->LBRCB.W. The file open count, FCB_addr->FBOPN is initialized to 0, and the system CCB pointer contains the address of the root CCB.

One LCB is allocated for the LDU. The LCB contains the data that LDU Management needs to remember with respect the LDU as a whole. Although none of the individual disk unit files that compose the LDU are accessible while the LDU is initialized, no FCB/CCB pair exists for them. However, one UDB is allocated for each. Unit Management provides a service, GTUDB.P, which allocates a UDB. Unit Management links the UDB to the device's DCT physical unit list (DCT_addr->DCPUL.W). If a process attempts to open a disk unit that is already open or is part of an initialized LDU, its UDB will be found on DCT_addr->DCPUL.W, and a "Device in use" error will result. LDU Management links the UDB to the LCB physical unit list, LCB_addr->LBUDP.W. The following diagram illustrates how these databases are linked together for the LDU file:



The FCB of any file contains the LCB address of the LDU on which the file resides. When a user issues an I/O request, CCB Management retrieves the LCB address from the FCB, and uses it as a parameter to the Buffer Management routine that enqueues the buffer header to the proper UDB (NQBHR).

The LDU initialization procedure is performed by the XINIT.P service. XINIT.P is accessed by user processes via the ?XINIT system call. XINIT.P provides support for initialization of both mirrored and non-mirrored LDUs. Although AOS/VS 7.50 supports the ?INIT system call, DINIT.P (?INIT implementation) actually translates the ?INIT system call packet into the valid ?XINIT system call packet, and subsequently calls XINIT.P. ?INIT can be used to initialized non-mirrored LDUs only.

The only variant of XINIT.P is IXINIT.P. System initialization calls IDINIT.P (internal variant of DINIT.P) to initialize the logical disk unit for the SWAP and PAGE directories, :BOTH. IDINIT.P reformats the input packet and calls IXINIT.P to perform the initialization operations. AOS/VS 7.50 does not support mirroring on :BOTH; therefore, IDINIT.P is called instead of IXINIT.P.

The C-based algorithm of the XINIT.P service is illustrated below. Most of the primitive operations have not been pseudo-coded; however, the comments provide detailed descriptions of the internal sequence of events that occur during LDU initialization. Read the comments! The only input to XINIT.P is the system call packet address. Output values are returned in the packet.

```
XINIT.P (*user_pkt_addr)          /* Input: ?XINIT packet      */
{
/************************************************************
 * XINIT.P: Initialize an LDU.  First some preliminary stuff. *
 ************************************************************/

   call move_packet_params_to_stack (user_pkt_addr, &pkt_addr);
   call move_pu_list1_to_stack (user_pu_list1, &pu_list1);
   call validate_packet_params;

/************************************************************
 * Each unit in the LDU must be validated.  One LCB for the   *
 * LDU must be allocated, and one UDB for each unit must be   *
 * allocated and linked to the LCB as well.  The following    *
 * routine performs this and other necessary actions.         *
 * It is expanded in the pseudo-code following XINIT.P.       *
 ************************************************************/

   call validate_units_and_allocate_databases;


  /************************************************************
   * Done examining physical units in the LDU.                  *
   * But, all the previous work must be repeated for the        *
   * second image list if a mirrored LDU is being initted!      *
   * Another LCB and UDB chain must be created for the          *
   * mirror image of the LDU being initted.                     *
   * The above routine is called again to accomplish this.      *
   * (Stack data collected from the first image is saved so     *
   * stack can be modified by the second image's data.)         *
   ************************************************************/

   if (pkt_addr->?XINIT_PKT.COUNT == 2)      /* Mirrored LDU? */
     {                                       /* Yes!          */
     call validate_units_and_allocate_databases;

   /************************************************************
    * Validate all kinds of mirror information to ensure that *
    * these LDUs can indeed be mirrored.  This routine is      *
    * expanded and explained in full detail in Section 6.12.   *
    ************************************************************/

     call validate_mirror_for_LDU_init;  /* Validate mirror   */

     } /* end of extra work for mirrored LDU */
```

```
/*************************************************************
 * Open the LDU and link the LCB into the system LCB chain.  *
 *                                                           *
 * LNKLCB.P: Traverses UDB chain on LCB and sets each unit's*
 *   first/last logical addresses (UDB_addr->UDFAH,UDLAH.W).*
 *   Effectively "open" the LDU by allocating and initting   *
 *   an FCB and a system CCB.  A CPB is explicitly           *
 *   allocated here as well. The local dir hierarchy level   *
 *   of LDUs is 0 (FCB_addr->FBLVL). The DIB Funny FIB is    *
 *   moved into the FCB Funny FIB area, and remains resident*
 *   until the LDU is released.  The CCB use count is        *
 *   initialized to 1.  The system CCB address is stored at  *
 *   LCB_addr->LBRCB.W, providing the LCB->FCB->CCB link.    *
 *   Finally, the LCB is linked to the tail of the global    *
 *   LCB chain (.ELCB).                                      *
 *************************************************************/

   call LNKLCB.P;                    /* This is a big job!     */
                                     /* Almost done, now!      */

/*************************************************************
 * Read in the name and ACL blocks of the LDU.               *
 * RDNAC.P: Allocates memory for the LDU name and ACL, and   *
 *   reads the name and ACL blocks into the area.  Their     *
 *   logical disk addresses were saved on the stack by       *
 *   INITLD.P.                                               *
 *************************************************************/

   call RDNAC.P;                     /* Read LDU name/ACL blks */

/*************************************************************
 * Create an FNB/FIB for the LDU in the directory specified  *
 * in the caller's packet.                                   *
 *                                                           *
 * GFTLDU.P: Graft LDU into the existing dir hierarchy.      *
 *   Calls DRSLV.P to get the parent dir CCB address.        *
 *   The parent CCB use count is incremented.                *
 *   Calls VCREATE.P to create the FNB/FIB DDEs.  The FCB    *
 *     address is stored in the FIB!  The caller must have   *
 *     write or append access to the dir in which the LDU    *
 *     is being initted.                                     *
 *   Calls VSACL.P to create FAC and fill it with ACL.       *
 *   Calls PESTAC.P to verify that caller has owner access   *
 *   to the LDU's root directory.                            *
 *   Finally, calls CULCK.P to unlock the parent CCB,        *
 *   which also FLUSHES THE FUNNY FIB, causing the "LDU      *
 *   initialized" bit (IBSIN) in the DIB to get set!         *
 *   (IBSIN was set by LNKLCB.P, but FCB was not flushed.)   *
 *************************************************************/

   call GFTLDU.P;                    /* LDU is initialized!!!  */
```

```
  /************************************************************
   * If mirrored LDU, set time stamps in the MDB and move     *
   * the LDU name to the MDB (MDB_addr->MDLDN).               *
   ************************************************************/

     call SET_START (MUNSYNC, MDWSR);     /* Set MDB time stamp*/
     call move_LDU_name_to_MDB;           /* LDU name to MDB   */

  /************************************************************
   * Finally, release any memory still allocated and return.  *
   * (RDNAC.P allocated memory for the LDU name and ACL, and  *
   * GTLCB.P allocated memory for the Funny FIB.)             *
   ************************************************************/

     call RSMEM (&LDU_name_block);        /* Formalities,      */
     call RSMEM (&ACL_block);             /*   formalities,    */
     call RSMEM (&Funny_FIB_block);       /*     formalities!  */

     return;

} /* end of XINIT.P */

/************************************************************
 * This routine validates that each physical unit is indeed *
 * a member of the LDU, and that each PU is valid.  An LCB is *
 * allocated for the LDU, and a UDB is allocated for each PU. *
 * The BBT address is read and saved in the UDB, and all UDBs *
 * are linked both through the LCB and through the DCT.     *
 ************************************************************/

validate_units_and_allocate_databases();
{
   for (pu = 1; pu <= pu_list1->?PUL_PKT.COUNT; pu++)
     {
  /************************************************************
   * Open physical unit.                                      *
   * OPENPU.P calls File Open Services XEOPEN.P to exclusively*
   * open the PU.  An FCB and (system) CCB is created.  The   *
   * disk unit's DIB is also read in, via a call to BLKIN.    *
   * Only ?FDKU (disk unit) files can be initialized, else an *
   * "Illegal device name type" error will be returned.  The  *
   * caller must have execute access to the unit file, else   *
   * a "File access denied" error is returned.                *
   * An LCB is allocated temporarily, and later released.     *
   ************************************************************/

     call OPENPU.P;                     /* Open the physical unit */

  /************************************************************
   * Check disk format revision number.                       *
   * The disk rev number is found at DIB_addr->IBREV.         *
   * Valid revs are:                                          *
   *   SCPRV (3) - Any disk, except KISMET II, no ADEX area.  *
   *   SCKRV (4) - KISMET II disk, no ADEX area.              *
   *   SCREV (5) - Any disk with an allocated ADEX area.      *
   ************************************************************/

     call CKREV.P;                      /* Check disk rev number  */
```

```
/************************************************************
 * Verify that the current state of the PU is valid to      *
 * ensure that the LDU can be initialized.                  *
 * (The PU must actually be closed before returning errors.)*
 ************************************************************/

   if (DIB_addr->IBLDF & IBSIP != 0)
     return (Cannot_init_LDU_with_sync_in_progress);

   if (DIB_addr->IBLDF & IBSIN != 0)
     return (Cannot_init_LDU___Must_run_FIXUP);

/************************************************************
 * Check the mirror state of the PU.                        *
 * If the PU is not part of a mirror, verify that the caller*
 * is not attempting to init a mirrored LDU by specifying 2 *
 * image lists in the packet.  If the PU is part of a       *
 * mirror, and the caller only specified one image list in  *
 * the packet, he must also specify the OVERRIDE option, or *
 * he will get an "Incomplete mirror specified" error.  If  *
 * the caller is attempting to init a mirrored LDU, the     *
 * mirror status in the DIB must show synchronized (MSYNC), *
 * or he will get "Mirrored LDU is not synchronized" error. *
 ************************************************************/

   call CKMIR;                          /* Check PU mirror state  */

/************************************************************
 * The following is only done for the first PU in the LDU!  *
 *                                                          *
 * GTLCB.P: allocates an LCB for the LDU and inits it,      *
 *   allocates a Bit Map FCB and inits it,                  *
 *   checks num of PUs in LDU is valid (DIB_addr->IBNPU     *
 *   must be between 1 and 8).  Also saves the DIB Funny    *
 *   FIB in memory for later, when it is stored in the FCB. *
 *                                                          *
 * SETFX.P: sets the "FIXUP recommended on this LDU" bit in *
 *   the LCB and in the caller's packet if the IBFXR bit is *
 *   set in the DIB flag word.  The init will be successful,*
 *   but the caller should release it and run FIXUP.        *
 *                                                          *
 * For all other PUs, check that the LDU unique ID in the   *
 * DIB is equal to that of the first unit's unique ID,      *
 * (which was previously saved on the stack).               *
 * The unique ID is found at DIB_addr->IBIDH,IBIDM,IBIDL.   *
 ************************************************************/

   if (pu == 1)
     {                                  /* For first PU in LDU... */
     call GTLCB.P;                      /* Get LCB, Bit Map FCB   */
     call SETFX.P;                      /* Test FIXUP bit in DIB  */
     }
   else                                 /* For all other PUs..... */
     call CKLDI.P;                      /* Validate LDU IDs.      */
```

```
/*************************************************************
 * Init the LCB and the Bit Map FCB.                         *
 * INITLD.P: moves current/max space from DIB to LCB, where  *
 *   it is maintained while the LDU is initted; inits Bit    *
 *   Map FCB params to start free block search at beginning  *
 *   of the LDU Bit Map.                                     *
 *************************************************************/

  call INITLD.P;                    /* Init LCB and BM FCB    */

/*************************************************************
 * Release and flush the system buffer containing the DIB.   *
 * An error will signal that there were problems flushing    *
 * the DIB.  For example, if the caller does not have write  *
 * access to the PU, an error will occur.  Therefore, this   *
 * flush is an access validation as well.                    *
 *************************************************************/

  call FLDIB.P;                     /* Flush DIB to disk      */
  if (error)                        /* Did it go?             */
    return (File_access_denied); /* No! Must assume ERFAD. */

/*************************************************************
 * Close the physical unit.                                  *
 * ECLOSE.P will perform the internal close of the PU that   *
 * OPENPU.P did earlier.  The FCB and (system) CCB will      *
 * disappear.  So will the LCB temporarily created by the    *
 * unit open.  The close is done now because no more I/O     *
 * will be done to this PU.                                  *
 *************************************************************/

  call ECLOSE.P (CCB_addr);         /* Close the PU now       */

/*************************************************************
 * Get a UDB for the PU.                                     *
 * A UDB is allocated from GSMEM.  If metering is enabled,   *
 * a metering area is allocated as well.  The UDB is linked  *
 * to the DCT PU list (DCT_addr->DCPUL.W).  A call is made   *
 * to the disk init routine (DCT_addr->DCTIU.W) to init the  *
 * UDB (head, cylinder, size info) and to load the data      *
 * channel map slots.  A disk recalibration (RECAL) command  *
 * is issued to the drive as well.                           *
 *************************************************************/

  call GDUDB.P;                     /* Allocate UDB, init unit*/

/*************************************************************
 * Read the Bad Block Table (phys addr in DIB).              *
 * If there are bad blocks, allocate one block of memory     *
 * and move the BBT data into the block.  The BBT remains    *
 * resident for as long as the LDU is initialized.  The      *
 * BBT pointer is stored at UDB_addr->UDBBT.W.  (There is    *
 * special handling for KISMET II type disks.  The BBT is    *
 * accessed by two pointers in the DCT: DCBB1.W, DCBB2.W.)   *
 *************************************************************/

  call RDBBT.P;                     /* Read, save BBT in mem  */
```

```
  /*************************************************************
   * Link the UDB to the UDB list at LCB_addr->LBUDP.W.        *
   * UDBs are linked in order of sequence number in the LDU.   *
   *************************************************************/

     call LNKUDB.P;                    /* Link UDB on LCB chain  */

     } /* end of for loop */
} /* end of validate_units_and_allocate_databases */
```

6.4.1 Special Case: Master LDU Initialization

System Initialization calls a special LDU Management service, MLDUI, to initialize the system master root LDU. The basic LDU initialization algorithm in MLDUI is identical to that of XINIT.P, however, some subtle internal changes are evident.

One such difference is that there is no system call packet associated with MLDUI. The physical unit list for the master LDU is dynamically built by SYSBOOT. SYSBOOT moves this list to the location in memory designated by AOS/VS label MLDTB. When MLDUI is invoked, it has direct access to the physical unit list required to boot the LDU. MLDUI validates these entries by comparing them to the device names and unit numbers in the Unit Table (UNTTB) that VSGEN built. If the master root is a (synchronized) mirrored LDU, SYSBOOT moves the physical unit list of the secondary image to the location in memory designated by the label MLDMIRTB.

Another difference is the assigning of a system buffer to read in the DIB. MLDUI must call ASBUF, the Buffer Management service that assigns the caller a system buffer, but takes the error return if no system buffer is available. XINIT.P calls BLASB to assign the buffer, which pends the caller's CB if no buffer is available. Although in the case of MLDUI the occurrence of such an event is highly unlikely, MLDUI cannot call BLASB, which would hang the system if a buffer could not be assigned.

Finally, if the master root is being initialized, the system CCB for the LDU is not allocated dynamically from GSMEM. The master root CCB is always found at AOS/VS label RTCCB. Furthermore, the master root has no parent directory. Therefore, the parent directory CCB pointer in the LDU's CCB and FCB is 0.

MLDUI takes the liberty of saving the master root LDU name at the address pointed to by global label MLDPT.W. MLDUI prints out the "Master LDU: ldu_name" message to the operator console as well.

6.5 LDU Release

When an LDU is initialized into the system, its root directory and all subordinate directories are available for use by system users (provided they have access privileges). The use count in the system CCB is incremented whenever Resolution Services opens the LDU, which occurs each time a file within the LDU is opened. The LDU is visible to the user community until it is "released."

LDU Release is essentially equivalent to deleting the LDU's directory data entries in its parent directory and then "closing" the LDU. The LDU can be released only if the use count in the system CCB is 1, the initial value stored by LDU Initialization. LDU Release must deallocate the disk-based and in-core databases that LDU Initialization allocated.

The FCB of any file contains the LCB address of the LDU on which the file resides. Before an FCB is allocated, the FFCB File Management operation checks to see if it already exists by examining the FIB. If the FCB memory address stored in the FIB (FIB_addr->FIFCB.W) is non-zero, the FCB is assumed to be resident. However, File Management knows that there is the slight possibility that the FCB may be invalid. For example, if the system came down without closing the file and was rebooted without running FIXUP over the LDU, an invalid FCB address would be found in the FIB. Therefore, FFCB checks that the existent FCB found at FIB_addr->FIFCB.W is:

1) in ring 0
2) is a valid ring 0 address
3) matches the FIB pointer in the parent directory

If any of these conditions evaluates false, and FFCB was called by File Deletion Services, FFCB will set the "FIXUP recommended on this LDU" bit in the LCB. If any of these conditions evaluates false, and FFCB was called by any other component, FFCB will set the "Must run FIXUP on this LDU" bit in the LCB. FIXUP is only recommended if the caller is File Deletion Services because no FCB buffers will be flushed to disk. In other cases, such as opens or closes, FCB buffers will later be flushed to disk and possibly corrupt files.

When an LDU is released, various validation checks must be made on the state of the LDU. For example, if the system is shutting down, the bits set by FFCB will be checked and if they are set, force an abnormal shutdown. LDU Release and File System Shutdown code may set the following bit positions in global location AOSBT. If AOS/VS System Shutdown detects that any one of these bits is set, it prints "Abnormal system shutdown" and various FIXUP messages:

ABNUCB (0) - Use counts too high for at least one root CCB.

ABNFOB (1) - Too many FCBs left when the Master Root LDU is being released.

ABNBFA (2) - Invalid FCB address found in FIB.

ABNNET (3) - Error occured from KCCB.P when releasing :NET CID.

There are three variants of LDU Release. They are represented by the following entry points:

Variant	Function
DRLSE.P	Release an LDU for a user process. Implementation of the ?RELEASE system call.
IRLSE.P	Internal release of an LDU.
IRLSNR	Internal release of an LDU, but do not release the LDU's FCB.

DRLSE.P is called by any user process to release a previously initialized LDU. The only restriction is that the caller must have write and execute access to the parent directory of the LDU.

IRLSE.P is called by AOS/VS System Shutdown and Emergency Shutdown code. Emergency Shutdown is run in two cases:

1) if the system panics, and
2) if the system manager issues the "START 50" command from the SCP-CLI (on a halted system).

ESD must first close all files that are open (i.e., files with a corresponding FCB). If the file type of a file to be closed is ?FLDU, ESD calls IRLSE.P to release the LDU instead of KFCB.P/KCCB.P. The last file to be closed is the master root LDU (:), and ESD calls IRLSE.P normally to release the LDU. ESD never returns from IRLSE.P when the master LDU is released.

Normal system shutdown is run when PID 2 (the OP CLI) terminates. Process Management checks for this condition, and if PID 2 is indeed terminating, system shutdown begins. The module that processes normal system shutdown is SDOWN.P. First, all processes are terminated. As a result, all files which had been opened by the processes are closed. Finally, SDOWN.P traverses the global LCB chain (.MLCB) and releases all initialized LDUs from the system. If the LDU's use count is 1, all files in the LDU have been closed normally by SDOWN.P, and the LDU can be released normally via a call to IRLSE.P. If the LDU's use count is greater than one, one or more files remain open, and the LDU is released abnormally through IRLSNR. This variant of LDU Release does not release the FCB. Furthermore, the "LDU is initialized" bit in the DIB is left set, prohibiting the LDU from being initialized without having FIXUP run. A global count of "abnormal" LDUs, NRFCB, is incremented, which will force an abnormal system shutdown when the master root LDU is released.

Release code has built-in tests to accommodate some special conditions that must be checked if the master root LDU is being released. If the releases of all the other LDUs in the system had been successful, IRLSE.P prints "System shutdown" at the operator console; otherwise, it prints "Abnormal system shutdown." IRLSE.P prints FIXUP messages as well.

The only input to LDU Release code is the pathname of the LDU to be released. The C-based pseudo-code algorithm of DRLSE.P (including tests made by its variants) follows:

```
DRLSE.P (*pathname)                    /* Input: BP to LDU pathname */
{
/*************************************************************
 * DRLSE.P: Release an LDU.                                  *
 * First retrieve the CCB of the LDU to be released.  DRSLV.P *
 * returns it LOCKED.                                        *
 * If the file is not of type LDU, it cannot be released.    *
 * If the use count is not 1 (initted value by XINIT.P), the *
 * LDU is still in use and cannot be released.               *
 *************************************************************/

   call DRSLV.P (*pathname, NSERULES+CKCOMLOG, 0, CCB_addr);

   if (CCB_addr->CBFCB.W->FBTYP != ?FLDU)  /* Is this an LDU? */
     return (Illegal_file_type);           /* Nope.  Sorry!   */

   if (CCB_addr->CBUSC != 1)               /* Is LDU in use?  */
     return (LDU_in_use);                  /* Yup.  Sorry!    */

/*************************************************************
 * If the master root LDU is being released, check if the    *
 * "abnormal shutdown" bit should be set.  If any other LDUs *
 * were released through IRLSNR (abnormal release: release   *
 * LDU but do not release FCB), NRFCB was incremented.       *
 * Therefore, if FCBCN-NRFCB is not equal to 1, some other   *
 * LDU was released abnormally, and the "abnormal shutdown"  *
 * bit must be set so that ESD will print out the message.   *
 *************************************************************/

   if (CCB_addr = RTCCB)                 /* Releasing Master LDU? */
     {                                   /* Yes.                  */
     if ( (FCBCN-NRFCB) != 1 )           /* All other LDUs rlsed  */
       set_bit (&AOSBT, ABNFOB);         /* released normally? NO!*/
     }

/*************************************************************
 * If not releasing Master LDU (if DRLSE.P), the caller must *
 * have write access to the LDU parent dir to release the LDU.*
 * Then, the LDU file's DDEs (FNB, FIB, FAC) must be deleted  *
 * from its parent dir.   NOTE: If Resolution Services tried *
 * to access (lock) the LDU's CCB while we had it locked, the *
 * caller must be unpended now.  Resolution Services does not *
 * try to get the lock again immediately, but rechecks the   *
 * existence of the filename (FNB) first.  Since we have     *
 * just deleted it, the FNB will not be found and Resolution *
 * Services will return ("File does not exist") to its caller.*
 *************************************************************/
```

```
  else                                    /* Not releasing     */
    {                                     /* Master LDU.       */
    call CLOCK (CCB_addr->CBPCB.W);       /* Lock parent CCB.  */

    call ESTAC.P (CCB_addr->CBPCB.W, &ACL_privs);
    if (! ACL_privs & write_access)
      return (File_access_denied);

    call VDELETE.P (CCB_addr->CBFIB, CCB_addr->CBPCB.W);

    if ( check_bit (CCB_addr, BCBPL) )  /* Unpend waiters of */
      call UNPEND (CCB_addr);           /* CCB lock.         */
    }

/*************************************************************
 * Release the modified BHs on the FCB buffer list.          *
 * Release the modified BHs on the Bit Map FCB buffer list.  *
 *************************************************************/

  call RELBF (FCB_addr);                /* Flush FCB and Bit */
  call RELBF (LCB_addr->LBMFC.W);       /* Map FCB buffers   */

/*************************************************************
 * Read the DIB (physical block 3 on first unit in LDU).     *
 * Update the DIB by moving (ECLIPSE/MV WBLM instruction) the *
 * contents of the FCB Funny FIB to the DIB buffer.          *
 *************************************************************/

  call BLKIN (CCB_addr, &DIB_BH_addr);  /* Read the DIB      */
  DIB_addr = DIB_BH_addr->BQADR.W;      /* Save DIB addr     */

  wblm (DIB_addr->IBFFB, FCB_addr->FBSTS, FCOML);

/*************************************************************
 * If the "must run FIXUP" bit is NOT set in the LCB, clear  *
 * the "LDU is initialized" bit in the flag word.  But, if the*
 * "FIXUP is recommended" bit is set in the LCB, set it in the*
 * DIB as well.  Then update the space fields in the DIB.    *
 *************************************************************/

  if (!check_bit (LCB_addr, BLBFX) )
    {
    DIB_addr->IBLDF = DIB_addr->IBLDF & (-IBSIN-1);
    if ( check_bit (LCB_addr, BLBFR) )
      DIB_addr->IBLDF = DIB_addr->IBLDF | IBFXR;
    }

  DIB_addr->IBCSH.W = FCB_addr->FBCPB.W->CPCSH.W; /* Update */
  DIB_addr->IBMSH.W = FCB_addr->FBCPB.W->CPMSH.W; /* space  */

/*************************************************************
 * Flush the DIB.                                            *
 * For mirrored LDUs, UPDMIR.P is called to write the DIB of *
 * each disk independently, since certain words in the DIB   *
 * must be unique for a mirrored LDU (LDU ID, other mir info).*
 * For non-mirrored LDUs, set the modified bit in the buffer *
 * header and call NQBH1 to write out the DIB.               *
 *************************************************************/
```

```
   if ( check_bit (LCB_addr->LBUDP.W,BUDMIR) ) /* Mirror LDU? */
     {                                         /* Yes.        */
     call UPDMIR.P;                            /* Write DIBs. */
     }
   else
     {                                         /* Not mirror! */
     set_bit (DIB_BH_addr, BQTMD);             /* Write DIB.  */
     call NQBH1 (&PPBSY, LCB_addr->LBUDP.W, DIB_BH_addr);
     call BWAIT;
     }


/*****************************************************************
 * Release the DIB buffer header.                                *
 * Release the Bit Map FCB memory.                               *
 * Release each UDB in the LDU.  Release the CPB memory.         *
 *****************************************************************/

   call RELD (DIB_BH_addr);                /* Release DIB BH  */
   call RSMEM (LCB_addr->LBMFC.W, FCBLT);  /* Release BM FCB  */

   for (each UDB in the LDU)               /* Release UDBs    */
     {
     if (! check_bit (UDB_addr, BUDMIR) )  /* Recal PUs in    */
       call RECAL (UDB_addr);              /* non-mirrored LDU*/

     call RLUDB.P (UDB_addr);              /* Release the UDB */
     }

   call RSMEM (FCB_addr->FBCPB.W, CPBLT);  /* Release the CPB */

/*****************************************************************
 * Do not release the FCB if this release is an IRLSNR, or if    *
 * there was an error writing to the DIB.  When the master       *
 * LDU is released, an abnormal shutdown condition will be       *
 * detected (FCBCN-NRFCB will not be equal to 1).                *
 * Otherwise, release the LDU's FCB.                             *
 *****************************************************************/

   if (IRLSNR || error_writing_the_DIB)
      do not release FCB;
   else
     call RFCB (FCB_addr);


/*****************************************************************
 * If the Master LDU is being released, do the following:        *
 * clear the LCB chain, disable interrupts, print appropriate    *
 * messages to the operator console.  Then continue with         *
 * system shutdown.                                              *
 *****************************************************************/

   if (CCB_addr = &RTCCB)                  /* Master LDU?     */
     {                                     /* Yes.            */
     .MLCB.W = -1;                         /* Clear out global*/
     .ELCB.W = -1;                         /* LCB chain.      */

     INTDS;                                /* Disable interpts*/
```

```
    if (AOSBT != 0)                          /* Abnormal shutdn?*/
      {                                      /* Yes.            */
      print ("Abnormal system shutdown");
      print_fixup_messages;
      }
    else                                     /* No.             */
      print ("System shutdown");             /* Normal shutdown.*/

    if (SHUTR == 0)      /* Did OP CLI trap?  No, continue    */
      goto STOP1;        /* normal shutdown in module ESD.SR. */
    else                 /* Yes, panic the system with code   */
      goto SHUTPR;       /* 10001, because the OP CLI has     */
                         /* terminated abnormally.            */
    }

/*************************************************************
 * Not releasing Master LDU.                                 *
 * Save the FIXUP error code (from LCB) if one is to be      *
 * returned to the caller.                                   *
 * Then destroy buffers on the LCB cache buffer list, remove *
 * the LCB from the global LCB chain (.MLCB), and release    *
 * the LCB memory.                                           *
 *************************************************************/

  saved_error_code = 0;
  if ( check_bit (LCB_addr,BLBFX) && !SHTDN )
    saved_error_code = "Must_run_FIXUP_on_LDU";
  else if ( check_bit (LCB_addr,BLBFR) && !SHTDN )
    saved_error_code = "Fixup_recommended_on_LDU";

  call CLRLCB (LCB_addr);                    /* Clear LCB buffs */
  call REMLV  (LCB_addr);                    /* Remove LCB      */
  call RSMEM  (LCB_addr, LBBLT);             /* Release LCB mem */

  if (mirrored LDU)                          /* If mirrored LDU,*/
    call RELMDB.P;                           /* rel MDB, MURBs  */

  call RSMEM (CCB_addr, CBBLT);              /* Release LDU's   */
                                             /* system CCB now. */

/*************************************************************
 * Finally, lock the parent CCB before decrementing its use  *
 * count twice: once for the LDU's released FCB, and once for *
 * the LDU's released CCB.  KCCBE.P unlocks the parent CCB    *
 * and releases the CCB hierarchy opened by DRSLV.P earlier   *
 * in this module.                                           *
 *************************************************************/

  call CLOCK (PCCB_addr);                    /* Lock parent CCB */
  PCCB_addr->CBUSC -= 2;                     /* Decr use count  */

  goto KCCBE.P (PCCB_addr, saved_error_code);   /* Good-bye! */

} /* end of DRLSE.P */
```

6.6 Bit Map FCB Parameter Definitions

There is one Bit Map FCB per LDU. Although this database is officially called an FCB, it contains only a few common parameters with a typical file's FCB, namely the LCB address (FBLCB.W) and the Buffer List Pointer queue descriptor (FBBLP.W). The Bit Map FCB is dynamically allocated from main system memory (GSMEM) during LDU Initialization, and its pointer is stored in the LCB at LCB_addr->LBMFC.W. The Bit Map FCB holds information used by the LDU Withdraw/Deposit Blocks operations. The most important parameters will be discussed in detail in the next section.

Offset		Bit Map File Control Block (Bit Map FCB)
FBLCB.W	0	LCB address.
FBBLP.W	2	Bit Map FCB Buffer List pointer.
FBSTS	6	Unused.
FBTNB.W	10	Total Number of disk blocks being requested.
FBLTE	11	Num blocks left in Bit Map from point of search.
FBBTLS	12	Original bit position of the free block.
FBLNAD.W	13	Logical disk address (lda) of next Bit Map block to read in.
FBBTLF	15	Number of disk blocks left to find.
FBNMB	16	Total number of disk blocks in Bit Map.
FBLFAD.W	20	First Logical Disk Address of Bit Map.
FBWLTS	21	Count of doublewords (or bytes) in left in current map block. If 0, the current block has been exhausted and the next must be read in.
FBFLG1	22	If 0, search for <= 14. blocks. If -1, search for > than 14. blocks.
FBBLKD	23	Relative block number in Bit Map of root block.
FBBITD	24	Root bit displacement.
FBTMP1.W	25	Temporary variable for WDCBK only.
BMF1.W	27	Lda of first Bit Map block where a single block was found, or where n blocks were deposited.
BMFN.W	31	Lda of first Bit map block where n blocks were found, or where n blocks were deposited.
BMPRN.W	33	Number of blocks (> 1) last allocated.
	35	Unused area.
	43	Unused area.
FBNBZN	44	Total number of zero bytes to be find (WDCBK).
FBNBZL	45	Number of zero bytes left to be found (WDCBK). (The rest of the space is unused.)
FCBLT	56	Length of Bit Map FCB.

6.7 The LDU Bit Map

The DFMTR utility allocates an LDU Bit Map when it creates a logical disk. Each bit in the LDU Bit Map indicates whether a corresponding logical disk block is free or in use. Bit 0 in the Bit Map corresponds to logical block 0; bit 1 corresponds to logical block 1; and so forth until the last logical disk address in the LDU. A zero (reset) bit indicates that the disk block is free; a one (set) bit indicates that the block is already in use. If a disk block is free, it can be allocated by setting the bit in the LDU Bit Map, thus "withdrawn" from the Bit Map. If a disk block is in use, it can be freed by resetting the bit in the LDU Bit Map, thus "deposited" to the Bit Map. The "root block" is the Bit Map block that contains the first 0 bit that satisfies the request. This terminology is used throughout AOS/VS LDU Management code.

The primary concern of AOS/VS LDU Management is to keep disk fragmentation to a minimum, even at the cost of increased Bit Map search time. One way to keep a sparse disk as compact as possible is to search the Bit Map from the beginning each time disk blocks are to be allocated. This will take the space closest to the beginning of the disk. However, as the disk becomes more full, unnecessary and wasteful time will be spent searching through Bit Map blocks that have no space available. To address this problem, while retaining the desirable feature of compacting new allocations to the beginning of the disk, the Bit Map FCB is used to maintain variables that optimize LDU disk block allocation.

BMFCB_addr->BMF1.W contains the logical disk address of the Bit Map block where the last single block was found, or where the last blocks were deposited. Thus, the search for a withdraw request of a single block begins at this block. If a free block is found, BMF1.W remains unchanged. If a free block is not found, each Bit Map block following BMF1.W is searched until the block is found. Then, BMF1.W is updated to contain the logical disk address of the new Bit Map block. When a single block is deposited, BMF1.W is examined. If the receiving Bit Map block occurs earlier in the Bit Map than BMF1.W, then BMF1.W is updated to point to the earlier block. Otherwise, it remains unchanged. BMF1.W is initialized to the logical disk address of the first Bit Map block.

BMFCB_addr->BMFN.W contains the logical disk address of the Bit Map block where the last block of size BMFCB_addr->BMPRN.W was withdrawn or deposited. The search for a withdraw request of n blocks, in which n < BMPRN.W, begins at map block BMF1.W. If n >= BMPRN.W, the search begins at BMFN.W. If the n blocks are not found, each following Bit Map block is searched until the n blocks are found. When the n blocks are allocated, BMFN.W is updated with the logical disk address of the beginning map block containing the n blocks (disk block allocation can cover multiple map blocks) and BMPRN.W is changed to n, regardless of the beginning point of the search. Therefore, BMPRN.W varies depending on the size of withdraw requests. When n blocks are deposited, the receiving Bit Map block is checked. If it occurs

in the same or a later block than BMFN.W, both BMFN.W and BMPRN.W remain unchanged. If it occurs at or before BMF1.W, both BMF1.W and BMFN.W are updated to point to the earlier block, but BMPRN.W is untouched because the next withdraw request search, regardless of its size, will begin at the same Bit Map block. Finally, if the n blocks are deposited somewhere between BMF1.W and BMFN.W, BMFN.W is reset back to BMF1.W, and BMPRN.W is again untouched. BMFN.W is initialized to BMF1.W and BMPRN.W is initialized to 1.

LDU Management reads Bit Map blocks into system buffers by calling the CCB Request Management service MBLKN. This specially provided service takes as an input the logical disk address of the block to be read. MBLKN calls Buffer Management (ASBUF) to assign a buffer. MBLKN then enqueues the buffer to the Bit Map FCB Buffer List queue and reads the specified Bit Map block into the buffer. Of course, the buffer may already be enqueued to the Bit Map FCB Buffer List queue, in which case no disk I/O would be necessary. The buffer header address is returned to LDU Management disk block withdraw and deposit routines. These routines release the system buffers modified (RELM). The modified buffers remain enqueued to the Bit Map FCB until the LDU it represents is released. This approach reduces the amount of total disk I/O and still provides full functionality.

6.8 LDU Disk Block Allocation (Withdraw Blocks)

There are two LDU Management operations that withdraw disk blocks from the LDU Bit Map:

1) WDBLK: withdraws 1 disk block, and
2) WDCBK: withdraws n contiguous disk blocks (n > 1).

These operations are called by CCB Request Management command processing routines. Upon being invoked, these operations must save the return address of the caller in the IOCB at save level 1 (IOCB_addr->IOSL1.W). Control is returned to the caller through an indirect jump through this location.

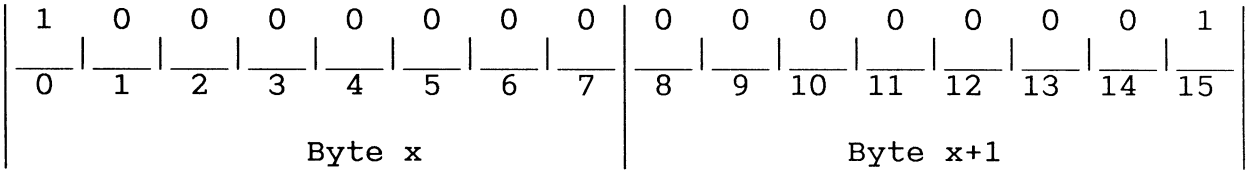
The number of blocks requested to be withdrawn for WDBLK is always 1. These requests are made to allocate file index blocks, data blocks in files with data element size 1, and directory bit map blocks. The number of blocks requested to be withdrawn for WDCBK is equivalent to the data element size of the file for which disk blocks are being allocated. This value is readily available from IOCB_addr->IODEH.W.

LDU Bit Map withdraws are done in two parts. First, a string of enough zero bits to satisfy the withdraw request must be located in the Bit Map. Second, the bits must be set. The zero bits cannot be set as soon as they are detected because LDU Management must ensure that enough free contiguous blocks will be found in the string first. Otherwise, too much inefficient and messy bit manipulation would result. For example, consider a withdraw request of 256. contiguous disk blocks. If only 250. contiguous blocks were found, and the insufficient number of zero bits spanned two map blocks, the first map block would have

to be read again and the bits reset. This algorithm would cause unacceptable performance and undue confusion.

The LDU Bit Map is searched in two ways: bit by bit if the withdraw request is between 1 and 14. blocks, or by byte if the withdraw request is greater than 14. blocks. If the request is for greater than 14. blocks, at least one full byte of zero bits (beginning on a byte boundary) must be found in the Bit Map. Therefore, the search for the specified number of blocks begins on a byte boundary, and full bytes are checked for zero values. If the request is for less than 14. blocks, it is possible that one full byte of zeroes is not needed to satisfy the request. Since the high seven bits of byte x are adjacent to the low seven bits of byte x+1, the 14. contiguous disk blocks to which they correspond could be allocated without filling up all 8 bits in either byte. Therefore, the Bit Map search for less than 14. blocks is done by bit. Each doubleword after the beginning point of search is checked for the specified number of free bits. When the bits are finally set (in part 2 of the withdraw algorithm), bit map data is accessed in doubleword increments, regardless of the number of blocks requested, because all contiguous bits are already known.

The following diagram illustrates why one full byte is not needed for disk block allocation requests of 14 or less blocks. The sufficient number of zero bits can be found in bits 1-14 of a word. Any larger request would require that either byte consist of all zeroes. It is clear why LDU Management searches the Bit Map by byte for requests larger than 14 blocks.



Most withdraw requests are for less than 14 contiguous blocks if the system parameter to create files with a default elementsize of 4 is specified at VSGEN time. This is the case with most AOS/VS systems. The Bit Map is searched sequentially from the beginning point of search (BMFCB_addr->BMF1.W or BMFCB_addr->BMFN.W) until the correct number of contiguous zero bits are located or until Bit Map blocks are exhausted.

The algorithm for withdraw requests of greater than 14 blocks is slightly more complicated. The algorithmic approach taken by WDCBK can be expressed as follows:

```
WDCBK (IOCB_addr);
{
/*************************************************************
 * Lock the Bit Map.                                         *
 * Withdraw and deposit operations are exclusive.            *
 *************************************************************/

   call MWAIT;

/*************************************************************
 * Calculate the number of full bytes needed to find.        *
 * Subtract 7 from original request, so that after the total *
 * number of 0 bytes are found, only 7 to 14 bits remain.    *
 * This remainder can be split across two bytes!             *
 * Then, calculate the number of extra bits needed to find.  *
 *************************************************************/

   BMFCB_addr->FBNBZN = (BMFCB_addr->FBTNB.W - 7) / 8;
   extra_bits_needed =  7 + (BMFCB_addr->FBTNB.W % 8);

/*************************************************************
 * Search the Bit Map for enough contiguous zero bytes.      *
 * If a set bit is found, search restarts at next byte.      *
 * If Bit Map exhausted, not enough contiguous disk          *
 * blocks error is returned.                                 *
 * Note: Bit Map blocks are read into system buffers         *
 * (implicitly by "search") and enqueued to the FCB buffer   *
 * list when released.                                       *
 *************************************************************/

   if (BMFCB_addr->FBNBZN >= BMFCB_addr->BMPRN.W)
     search (BMFCB_addr->BMFN.W, bytes_needed, &exhausted)
   else
     search (BMFCB_addr->BMF1.W, bytes_needed, &exhausted)

   if (exhausted)
     {
     error_code = Not_enough_contiguous_disk_blocks;
     call IOWU;                    /* Unlock BM, unpend waiters */
     XJMP @(IOCB_addr->IOSL1.W);/* Error return to CCB Mgmt. */
     }

/*************************************************************
 * Enough bytes found.  Between 7 and 14 bits remain.        *
 * First check the doubleword previous to the root block.    *
 * If the extra bits are not found, check the address after  *
 * the last byte checked.  (Implicitly, if the byte is the   *
 * last one in the map block, the NEXT map block would be    *
 * read in.)                                                 *
 *************************************************************/
```

```
   backsearch (BMFCB_addr->FBTMP1,
               extra_bits_needed, &num_bits_found);

   if (num_bits_found < extra_bit_needed)
     search (LCB_addr->LBMBF.W->BQADR.W + current_displacement,
             extra_bits_needed-num_bits_found, &num_bits_found);

/************************************************************
 * Has the request been satisfied?                          *
 * The bits before the first full byte and after the last   *
 * byte have been checked.  If the extra bits have been     *
 * found, the requested number of free blocks can be        *
 * withdrawn from the bit map.  So, set the bits!           *
 ************************************************************/

   if (num_bits_found == extra_bits_needed)  /* Found enough?*/
     set_the_bits (BMFCB_addr->FBLNAD.W,     /* Start block  */
                   BMFCB_addr->FBBITD,       /* Bit offset   */
                   BMFCB_addr->FBTNB.W);     /* Num bits req */

   else                                      /* Not enough!    */
     continue_search;                        /* Keep searching */

/************************************************************
 * Update Bit Map n-block pointers.                         *
 * Unlock the Bit Map and wake up IOCB waiters.  Return.    *
 ************************************************************/

   BMFCB_addr->BMFN.W = BMFCB_addr->FBLNAD.W;
   BMFCB_addr->BMPRN.W = BMFCB_addr->FBTNB.W;

   call IOWU;                      /* Unlock BM, unpend waiters */
   XJMP @(++IOCB_addr->IOSL1.W);/* Good return to CCB Mgmt.  */

} /* end of WDCBK */
```

6.9 LDU Disk Block Deallocation (Deposit Blocks)

There are two LDU Management operations that deposit disk blocks from the LDU Bit Map:

1) DEBLK: deposits 1 disk block, and
2) DEBKS: deposits n contiguous disk blocks (n > 1).

Like the withdraw operations, these operations are called by CCB Request Management command processing routines. Since withdraws and deposits are exclusive operations (the Bit Map global lock is acquired during both), the return address of the DEBLK caller is saved at the same IOCB save level as the WDBLK caller, IOCB_addr->IOSL1.W. Control is returned to the caller through an indirect jump through this location.

The number of blocks requested to be deposited for DEBLK is always 1. These requests are made to deallocate file index blocks, data blocks in files with data element size 1, and directory bit map blocks. The number of blocks request to be withdrawn for DEBKS is equivalent to the data element size of the file for which disk blocks are being deallocated. The data element size is found at IOCB_addr->IODEH.

The C-based algorithm for DEBKS is illustrated below. The inputs are the logical disk address of the first block to deposit, and the IOCB address.

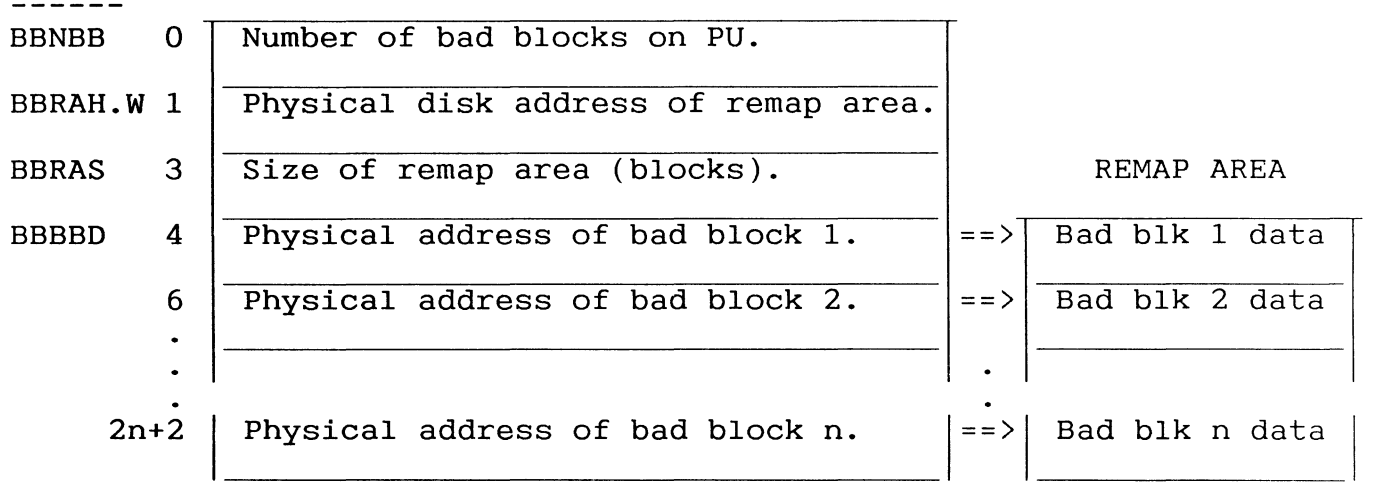
```
#define BITS_PER_BLOCK 4096
DEBKS (lda_first_block, IOCB_addr);
{
/*************************************************************
 * The LCB cache buffers of logical disk data (index blocks, *
 * DDBs, bit map blocks) must be removed since the data will *
 * become invalid when this operation completes.             *
 * The Bit Map lock must be acquired.                        *
 *************************************************************/

   call DELLCB;                    /* Buffer Mgmt service to    */
                                   /* remove BHs from LCB cache.*/
   call MSWAT;                     /* Acquire Bit Map glob lock.*/

/*************************************************************
 * Calculate the first relative Bit Map block to which the   *
 * first block to be deposited corresponds.                  *
 * Then adjust the 1-block and n-block pointers in the Bit   *
 * Map FCB. These conditions were discussed in Section 6.6.  *
 *************************************************************/

   BMFCB_addr->FBLNAD.W = BMFCB_addr->FBLFAD.W +
                          lda_first_block / BITS_PER_BLOCK;

/* This block falls before 1-block pointer?                  */
   if (BMFCB_addr->FBLNAD.W < BMFCB_addr->BMF1.W)
     {                                          /* Yes       */
     BMFCB_addr->BMF1.W = BMFCB_addr->FBLNAD.W; /* New BMF1 */
     BMFCB_addr->BMFN.W = BMFCB_addr->FBLNAD.W; /* New BMFN */
     }

/* This block falls before n-block pointer (but after BMF1)?*/
   if (BMFCB_addr->FBLNAD.W < BMFCB_addr->BMFN.W)
     {                                          /* Yes       */
     BMFCB_addr->BMFN.W = BMFCB_addr->BMF1.W;   /* New BMFN */
     }

/*************************************************************
 * The block falls after the current n-block pointer BMFN.W. *
 * Neither pointer is adjusted because n is larger than      *
 * PMPRN.W.  Anyway, now is the time to read the bit map and *
 * reset the bits corresponding to the logical disk blocks   *
 * being deallocated.                                        *
 *************************************************************/
```

```
   reset_the_bits (BMFCB_addr->FBLNAD.W,      /* Start block  */
                   BMFCB_addr->FBBITD,        /* Bit offset   */
                   BMFCB_addr->FBTNB.W);      /* Num bits req */

/*************************************************************
 * Time to go home.                                          *
 * Unpend the first IOCB waiting for the lock.  Then return. *
 *************************************************************/

   call IOWU;                        /* Unlock BM, unpend waiters */
   XJMP @(++IOCB_addr->IOSL1.W);/* Good return to CCB Mgmt.  */

} /* end of DEBKS */
```

6.10 Bad Block Remapping

The Bad Block Table (BBT) for Data General model disks, except on Model 6214 (KISMET II) disks, resides in physical block 2 of each physical unit, and is only one block in length. The address of the BBT for KISMET II type disks is found in the DIB, and occupies 4K bytes (8 disk blocks) of space. This larger BBT is necessary because KISMET II type disks typically have an abnormally high number of bad blocks. The remainder of this section will assume non-KISMET II disks.

The BBT contains the physical disk addresses of up to 126. "bad" blocks, sectors whose data cannot be transferred by the disk controller. Each LDU can have a maximum of (126. * number_of_physical_units) bad blocks. Bad blocks are flagged by the AOS/VS DFMTR utility, and their addresses are written to the BBT. In addition, the DFMTR allocates a remap area on the physical unit (in visible space), to which the data at the bad block is remapped. There is a one-to-one mapping between the bad blocks listed in the BBT and the good data for that block in the remap area. The BBT is set up as follows:



When an LDU is initialized, the BBT of each physical unit is read in and maintained memory resident for as long as the LDU remains initialized into the system. (Its memory address is found at UDB_addr->UDBBT.W.) Dynamic bad block remapping is not supported under AOS/VS 7.50. If a bad block that is not listed in the BBT is encountered by the disk driver, the particular data transfer will be aborted, and a message will be sent to the operator console. However, LDU Management provides a service to the disk drivers, which checks the physical disk addresses specified in a logical disk I/O request against those addresses listed in the BBT. If a request is found to contain a bad block listed in the BBT, its data in the remap area is accessed instead. This service is called REMAP.

Before examining the BBT, REMAP divides it into segments of 32. words. Since bad block addresses are written to the BBT consecutively (in ascending order), the segment containing the range of addresses to which the request corresponds is isolated. The search for bad blocks begins at this offset in the BBT. On the average this is faster than stepping through the BBT from bad block 1 to the end.

The reason for dividing the BBT into segments rather than using a binary search is the fact that I/O requests may be for more than one block. This means that the BBT is not searched for an exact match, but by the range of addresses specified by the request. Therefore, a binary search of the BBT would be inefficient and complicated.

The reason for choosing a segment size of 32. blocks is to minimize the impact on non-6214 disks (non-KISMET types), whose BBT's will typically involve only 1 or 2 segments. The 6214-type disks (KISMET types) allow a maximum bad block count of 1022. A segment size of 32. is reasonable. (This section assumes bad block remapping for non-KISMET type disks.)

Consider the following example. If the request specifications were to write 4 blocks beginning at physical address 04300, and the BBT flagged address 4301 at word offset 140. (segment 1), the bad block would be found right away. Furthermore, each BBT entry is compared only once to the first and last address of the request (4300 <= BBT entry <= 4303); it is not checked against all of the addresses (4300, 4301, 4302 and 4303). If the BBT entry falls within the range, REMAP knows that the bad block is part of the request.

If REMAP determines that there is a bad block within the disk area specified by the request, REMAP must break down the request, because only contiguous data can be transferred on one I/O command to the controller. Continuing from the example above, REMAP breaks down the single request into three separate requests as follows:

Original Request	Request Part 1	Request Part 2	Request Part 3
UDB_addr->UDNBK = 4	3	2	0
UDB_addr->UDTBK = 4	1	1	2
UDB_addr->UDDAH.W =	04300	04301	04302
UDB_addr->UDRMH.W =	04300	remap_area_addr + 17408.	04302

Upon receiving the parameters of the original request, REMAP examines the BBT and finds bad block 04301 at word offset 140.

first_addr (04300) <= BBT entry 04301 <= last_addr (04303)

Since the bad block is not the first block to be transferred in the request, all of the contiguous, "good" blocks previous to it must be transferred first. REMAP modifies UDB_addr->UDTBK such that the number of blocks to be transferred for "Request, Part 1" is 1. REMAP returns the correct address to access in UDB_addr->UDRMH.W, and from this location the disk driver retrieves the actual physical address of the disk-based data. Finally, upon returning from REMAP, the disk driver performs the following subtraction to determine whether the request is to be split up:

UDB_addr->UDNBK = UDB_addr->UDNBK - UDB_addr->UDTBK.

In this example, UDB_addr->UDNBK, the number of blocks remaining to be transferred is set to 3.

UDB_addr->UDNBK is checked by the interrupt service routine, and the non-zero value will indicate that another request (of 3 blocks) must be enqueued. (A zero value indicates that the request is complete, and the appropriate post-processor is invoked.) Consequently, REMAP will be called again with the updated physical disk address (UDB_addr->UDDAH.W == 04301). REMAP will repeat the same procedure, and this time find that the bad block address is the same as the first address of the request. REMAP modifies the UDB fields again such that the number of blocks to be transferred for "Request, Part 2" is 1. The disk driver sets the number of block to be transferred next time to 2. Since the bad block was found at BBT offset 140., the physical address of the bad block to be transferred is actually found at physical address:

remap_area_addr + [(140. - 4)/2 * 256.]
= remap_area_addr + 17408.

If the block at physical address 04302 were bad also, two blocks would be transferred in "Request, Part 2" beginning at the same remap area address.

When "Request, Part 2" is complete, the disk driver will again check UDB_addr->UDNBK. Since "Request, Part 2" was not able to complete the request because there were still two good blocks left, "Request, Part 3" must be enqueued. REMAP is called and determines that there are no more bad blocks in the request. It sets UDB_addr->UDTBK to 2 (the remaining number of blocks in the request). The disk driver subtracts UDNBK-UDTBK, which yields a zero result. Upon completion of "Request, Part 3," the interrupt service routine will check UDNBK and realize that the request is complete. Thus, the entire data transfer was executed, even though the disk-based data contained a bad block.

It must be emphasized that all disks may encounter undetected bad blocks during data transfers. If this happens, a hard error will be reported to the operator console and to the process that issued the I/O. As of AOS/VS 7.50, bad blocks are not remapped dynamically. The user must run a partial format on the LDU to declare the new bad block. DFMTR will enter this new block in the BBT. It is a restriction that physical blocks 0 - 7 must be good; they cannot be remapped.

6.11 Mirroring Functionality

6.11.1 Terminology

The reader must be familiar with the following terminology in order to comprehend the AOS/VS Mirroring discussion in this section. Some of the terms have already been presented in previous sections; however, they have been provided again here to consolidate the topic of mirroring.

MIRRORING - a method of maintaining two copies of the same data. If one copy of the data is unavailable, it can still be accessed from the other transparently to user applications.

LOGICAL DISK MIRRORING - a method of maintaining two or more copies of the same LDU. If one LDU image becomes unavailable, the LDU is still available on the remaining image. All LDU images must have the same LDU name.

LDU IMAGE - When an LDU is mirrored, each copy of the LDU data is called an image.

HARDWARE MIRRORING - mirroring functionality that is provided at the disk controller level. Hardware mirroring is only effective when mirroring LDU images that exist at identical physical block addresses of physical disks on the same controller.

SYNCHRONIZED MIRROR - A mirror is synchronized when all data on all images is identical. When in this state, all data is written to all images but may be read from any image.

UNSYNCHRONIZED MIRROR - A mirror is unsynchronized when one data image is not identical to the other. When in this state, all data is written to all images, but data is only read from the preferred image (as specified by the System Manager or user).

BROKEN MIRROR - A mirror is broken when one of the images becomes unavailable. A mirror can be broken by the operating system (in the case of a disk failure) or by the System Manager/User (for backup reasons). All data can be read/written only on the remaining image(s).

RESYNCHRONIZATION - The process used to transform an unsynchronized mirror to a synchronized mirror. The resynchronization insures that all data on all images is identical prior to placing the mirror in a synchronized state.

OUT-OF-PHASE MIRROR - A mirror is out of phase when the image selected by the operator/user as the preferred image is not the most recent image.

6.11.2 LDU Mirroring and the System Environment

Logical disk mirroring is supported in AOS/VS 7.50. This implementation requires hardware mirroring as provided by the ARGUS family of disk subsystems. Mirroring is only supported for the ARGUS family of disk drives with hardware mirroring support. This support requires that mirrored units reside on the same controller and that no more (or less) than two disks exist in a mirror.

Logical disk mirroring does not require support from any software products outside of those included in the operating system product, except for those user applications that perform ?INIT/?RELEASE system calls. ?INIT does not support logical disk mirroring; the new ?XINIT system call must be issued instead. User applications must change if they intend to use Logical Disk Mirroring functionality.

Logical disk mirroring has no specific hardware requirements other than adequate disk storage for two LDU images. The initial implementation of mirroring requires hardware mirroring as provided by the ARGUS family of disk subsystems (6236 class). This disk configuration must include at least one controller and two disk units. The initial logical disk mirroring implementation will not allow mirroring of more than two images (limited by hardware mirroring). The design allows for more images to be added in the future.

6.11.3 Mirroring and Performance Implications

Logical disk mirroring may improve performance when the ratio of read to write operations is high. Since read operations can be performed on any image, operations may be optimized so that data is read from the image that can access the data fastest.

Performance may degrade when the ratio of reads to writes is low. A write operation will complete only when the data has been written to both images. The time needed to complete the write operation includes seek and rotational latency for two disks rather than one. The overall performance depends on data placement on the physical disk and the read/write ratio.

Performance will also degrade when mirroring a high speed disk with a slower speed disk. The aggregate performance for write operations will slow down to the speed of the slower disk, and thus will exhibit the performance of the slower write.

The performance of each write also depends on the request queue length for the physical units containing each mirrored image. If a mirrored write occurs on two images, and one of the images resides on a "busy" disk, the write will not complete until the busy disk has serviced the write request. Disk optimization algorithms may need to be adjusted so that mirrored operations have greater priority. This issue has not been thoroughly analyzed.

Resynchronization of logical disk images also will degrade performance. During resynchronization, all allocated blocks on the "good" logical disk image must be copied to the "bad" image. This I/O degrades performance of all applications using the LDU that is being resynchronized.

6.11.4 Functional Overview of Logical Disk Mirroring

Logical disk mirroring allows the user to use identical logical disk images as one LDU. When one of the images becomes unavailable, the data is still available on the other logical disk image. Since logical mirroring implies that both images are logically equivalent copies of each other, all write operations must occur to both logical images. However, read operations may occur on either image, allowing the system to retrieve the data from the image that can access the data most rapidly.

Mirrored logical disks exist in a number of states. The states of a logical mirror are as follows:

SYNCHRONIZED: When all images are complete mirrors of each other. Data can be retrieved from any image, but write operations must be performed to all images.

BROKEN: A broken LDU exists when one of the images becomes unavailable. All subsequent reads and writes to the mirrored LDU go to the "good" image(s) only.

UNSYNCHRONIZED: An unsynchronized LDU exists when the mirror has been broken previously and one image must be resynchronized. Writes will go to all images, but reads may only come from the "good" image until the mirror is synchronized again. The "good" image is selected by the system manager/user based on information provided by the system.

In order to determine which state a mirror is in, a synchronization method is required. A mirrored logical disk created under AOS/VS 7.50 is considered synchronized when both logical disk images have identical synchronization keys, time stamps, and logical disk names. Additionally, identical bad block tables are required. It is also a requirement that both images have different LDU unique IDs.

The synchronization key is a 32-bit counter that is incremented whenever a state change occurs. The synchronization key is only incremented on one image when a mirrored LDU is initialized. This approach is used so that after a power failure or system panic, mirror inconsistencies can be detected. Effectively, then, only synchronized mirrored images may be initialized as a

mirrored LDU. When it is initialized, the mirror state is changed to "unsynchronized," and remains unsynchronized until the LDU is released. If the images are identical, the mirror state is changed back to "synchronized." The synchronization key is incremented when any of the following events occurs:

- The mirrored LDU image is opened (one image only).
- The mirrored LDU image is closed.
- A file system utility (that performs writes) accesses the mirrored LDU.
- The mirror is broken (the synchronization key on the "good" LDU image is incremented)

The time stamp is a 64-bit value consisting of the time of day and date as maintained in internal system format. The larger value indicates the most recent time. Since stand-alone file system utilities do not maintain time and date, this value is not used by those utilities. The time stamp is incremented when any of the following events occurs:

- The mirrored LDU image is closed.
- The mirror is broken (the time stamp on the "good" LDU image is updated).

Resynchronization must be performed after the preferred LDU image is initialized into the file system hierarchy. The logical disk name of both images must be the same in order to perform mirroring. The logical disk unique ID of both images must be different.

The bad block tables of both logical disk images must be identical. If the bad block tables are not the same for both logical disks, a partial format must be run over the mirrored disk and the bad block tables must be merged. This is a restriction of hardware mirroring.

The resynchronization process includes verifying the equivalence of the bad block tables, establishing an "out of sync" mirror relationship, copying the "good" disk to the "bad" disk, then establishing a synchronized mirror relationship.

6.12 Mirroring Internals

6.12.1 Internal Mirroring Databases

The system task that performs all mirror functions runs on two unique databases. Each mirrored LDU has one Mirror Descriptor Block (MDB). All MDBs are linked to the global MDB chain, MDBCHN.W. The MDB chain of Mirror Unit Request Blocks (MURB), consists of one MURB for each physical disk unit in the LDU. Each MURB represents a mirrored UDB, and the request running on it. Normal I/O to a mirrored unit does not run on a MURB. Only mirror state changes, and mirror synchronizations run on a MURB. Only when an action on each MURB has completed does the action get marked as completed in the MDB. Therefore, mirror state is maintained in the MDB and is not necessarily equivalent to the mirror state of each physical piece.

The MDB parameter definitions are described below:

Offset		Mirror Descriptor Block (MDB)
MDLNK.W	0	MDBCHN.W forward link.
MDSTATE	2	Current MDB state.
MDDAT.W	3	Date (sync record).
MDTOD.W	5	Time (sync record).
MDMST	7	Mirror state (sync record).
MDLCB.W	10	LCB address for this mirror.
MDID1	12	LDU unique ID for primary image.
MDID2	15	LDU unique ID for secondary image.
MDLDN	20	LDU name.
MDMRB.W	40	MURB chain.
MDNMRB	42	Number of MURBs on MURB chain.
MDCMRB	43	Number of completed MURBs.
MDBMU.W	44	Bit Map UDB address.
MDFLG	46	MDB flag word.
MDERR	47	Error word.
MDPTB.W	50	Process table address of ?MIRROR issuer.
MDTCB.W	52	User TCB address of ?MIRROR issuer.
MDLN	54	Length of MDB.

The MDB states stored at MDB_addr->MDSTATE consist of the following values:

MDIDL (0) - Idle
MDSDR (1) - Synchronize Disk Ready
MDSDP (2) - Synchronize Disk in Progress

MDBRM (3) - Break Mirror
MDDLM (4) - Delete Mirror
MDUSM (5) - Unsynchronize Mirror
MDSYM (6) - Synchronize Mirror
MDWSR (7) - Write Sync Record

The following bit positions are defined for the MDB flag word MDB_addr->MDFLG:

```
MDTRAN (0) - MP Lock
MDDUC  (1) - Don't Unpend Caller
MDABT  (2) - Abort Sync Request
MDEDS  (3) - Error During Sync
MDSWT  (4) - Switch
MDHDE  (5) - Hard Disk Error
MDLTE  (6) - Log This Event when complete
MDSTL  (7) - Event Ready to Log
MDUTC (10) - Unpend TCB
MDUPA (11) - Unpend Abort Request
MDABR (12) - Request Aborted by Release of Disk
MDABC (13) - Request Aborted by System Call
```

The parameter definitions for the MURB are described below:

Offset		Mirrored Unit Request Block (MURB)
MULNK.W	0	MURB chain forward link.
MUSTATE	2	Current MURB state.
MUUDB.W	3	Mirror UDB address.
MUMDB.W	5	MDB address.
MUSAD.W	7	Current sync address.
MULAD.W	11	Last sync address.
MUBAD.W	13	Bit Map buffer header address.
MUFAD.W	15	First Bit Map address.
MUDSK.W	17	Bit Map disk address.
MUBMO.W	21	Current Bit Map offset.
MUFLG	23	MURB flag word.
MUEBH	24	Start of embedded buffer header.
MULN	60	Length of MURB.

The MURB states stored at MURB_addr->MUSTATE consist of the following values:

MUIDL (0) - Idle
MUSDR (1) - Synchronize Disk Ready
MUSDP (2) - Synchronize Disk in Progress

MUBRM (3) - Break Mirror
MUDLM (4) - Delete Mirror
MUUSM (5) - Unsynchronize Mirror
MUSYM (6) - Synchronize Mirror
MUWSR (7) - Write Sync Record
MUWSP (10) - Write Sync Record in Progress

The following bit positions are defined for the MURB flag word MURB_addr->MUFLG:

MUTRAN (0) - MP Lock bit
MUABT (1) - Abort Sync Request bit

For each mirrored LDU, there exists one MDB. All the MDBs in the system are linked through the global MDB chain MDBCHN.W. For each mirrored unit in the LDU, there exists one MURB. The MURBs are linked through MDB_addr->MDMRB.W.

Because a mirrored LDU actually consists of two logical disks (one which is a mirror image of the primary image), there must exist a UDB for each physical unit in the LDU. A mirrored LDU consisting of one mirrored unit (i.e., two physical units) requires two UDBs. The UDBs indicate to the File System and to the driver world that both devices are indeed in use. The UDBs are linked through the DCT of the device to which they belong.

Despite the fact that a mirrored LDU actually consists of two logical disks, it represents a single LDU. Only one LDU name is grafted onto the directory hierarchy when the LDU is initialized. Therefore, there exists an additional UDB, called a "Mirror UDB," for each mirrored unit. The Mirror UDB is linked into the LCB of the mirrored LDU. The Mirror UDB is parametrically defined the same as the typical UDB, but contains three redefinitions:

Offset		Mirror UDB Parameter Redefinitions
UDDCT.W	0	DCT address of this device.
UDUNT	2	Device unit number.
		:
		:
UDPRM.W	47	Primary image UDB address.
UDSCM.W	51	Secondary image UDB address.
UDCMS	53	Current Mirror State
		:
		:

A sample configuration of a mirrored LDU aids in comprehending the somewhat abstract concept of mirroring. Consider the following devices genned into an AOS/VS 7.50 system, and the following mirrored LDU created. The diagram on the following page displays the presence and the linkage of the databases for the mirrored LDU:

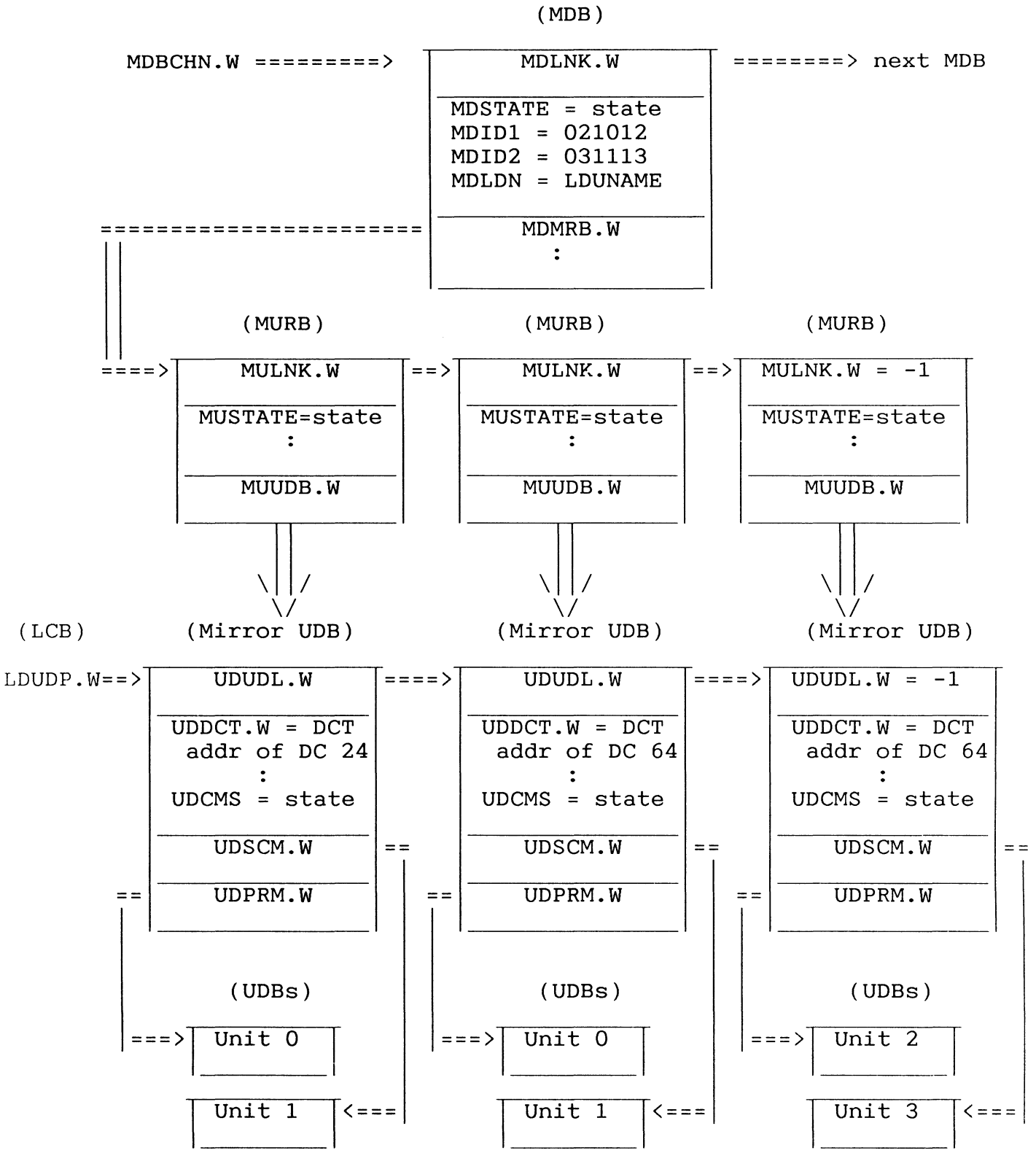
Devices:

DPJ<0,1>	DPJ1<0,1,2,3>
Device Code: 24	Device Code: 64

Mirrored LDU:

Primary Image ID: 021012
Primary Image Units: DPJ0, DPJ10, DPJ12

Secondary Image ID: 031113
Secondary Image Units: DPJ1, DPJ11, DPJ13


(MDB)
MDBCHN.W =========>
MDLNK.W
========> next MDB
MDSTATE = state
MDID1 = 021012
MDID2 = 031113
MDLDN = LDUNAME
MDMRB.W
(MURB)
(MURB)
(MURB)
MULNK.W
MULNK.W
MULNK.W = -1
MUSTATE=state
MUUDB.W
(LCB)
(Mirror UDB)
(Mirror UDB)
(Mirror UDB)
LDUDP.W==>
UDUDL.W
UDUDL.W = -1
UDDCT.W = DCT addr of DC 24
UDDCT.W = DCT addr of DC 64
UDCMS = state
UDSCM.W
UDPRM.W
(UDBs)
(UDBs)
(UDBs)
Unit 0
Unit 1
Unit 0
Unit 1
Unit 2
Unit 3

6.12.2 Running Mirror Requests

Mirror requests are run by the System Manager Task (SMTSK). In order to start a mirror request, the MDB state must be changed to perform the appropriate action, and the System Manager Task must be woken up. The System Manager Task walks down the MDB chain and looks for MDBs that are not in the idle state. The system manager task dispatches on the MDB state, and takes the appropriate actions.

Some MDB states require work to be done on individual units. This work is done using MURBs. MURBs allow a mirror operation to occur simultaneously on all physical units in the LDU. Synchronization of a mirrored LDU is the primary motivator behind the MURB/MDB design. When synchronizing an LDU, each physical unit will get synchronized simultaneously. The LDU will not be marked as synchronized until each piece is synchronized (the MURB complete count is maintained in the MDB).

Each MURB contains a buffer header that is used for the mirror request. Each buffer header is enqueued using a unique post-processor that awakes the System Manager Task upon completion.

The System Manager Tasks runs MDB requests. The MDB request just starts the MURBs running to put the mirrored LDU in the desired state. When all MURBs have completed, the MDB request is complete.

The ?MIRROR system call allows the user to mirror, synchronize, and remove LDU images. The LDU Management service that implements ?MIRROR requests is MIRROR.P. MIRROR.P accepts the LDU pathname as an argument, implying that the primary LDU image must already be initialized. When MIRROR.P is invoked, the following sequence of events occurs:

```
MIRROR.P ( )
{
   Validate ?MIRROR parameters;
   Check ACLs;
   Create UDBs for each unit in the secondary image;

   Ensure all LD Unique IDs match in the secondary image;
   Ensure all UDB info is correct;
   Ensure controllers support mirroring;
   Ensure mirrored pieces are on same controller;

   Check time stamps for FORCESYNC condition;

/* Now the mirror must be synchronized. */

   If (MDB does not exist for this LDU)
      {
      Create an MDB with MURB for mirrored unit;
      Init MDB/MURBs;
      Store LD unique ID of each image in MDB;

      Get a Mirror UDB for mirrored unit and init it;

      Set UNSYNCED state in MDB;

      Call SWAKE (unpends System Manager Task);
      Pend awaiting completion;

      If (error)
         break mirror;
         return (error);
      Else
         Create linked list of mirror UDBs;
         Link into LCB;
      }

   Else  /* MDB already exists for this LDU */
      {
      Link sechedary image UDB to mirror UDB;

      Set UNSYNCHED state in MDB;

      Call SWAKE (unpends System Manager Task);
      Pend awaiting completion;

      If (error)
         break mirror;
         return (error);
      }
```

```
   Set state to SYNC DISK READY in MDB;
   Set MDB "Don't Unpend Caller' bit if needed;

   Call SWAKE to start synchronization;
   Pend awaiting completion (if necessary);

   If (error)
      return (error);
   Else
      return;

} /* end of ?MIRROR */
```

The System Manager Task calls LDU Management service MIROP.P to run an MDB. An MDB is run to change the state of a mirror. MDB processing consists of running all MURBs on each physical unit in the mirrored LDU. Only when all the MURB processing is complete can the mirror state in the MDB be altered. MIROP.P calls the LDU Management operation RUNMURBS to run the MURBS in the mirrored LDU. The RUNMURBS algorithm is illustrated after MIROP.P. MDB processing is triggered either by initialization of a mirrored LDU or by issuing a ?MIRROR request. The algorithm of the sequence of events that occurs in MIROP.P during MDB processing are the following:

```
MIROP.P( )
{
   for (each MDB on MDBCHN.W)
     {
     Get the MDB state and dispatch;

     switch ( MDB_addr->MDSTATE )
        {
        /*****************************************************/
        IDLE:
        case MDIDL:
           break;

        /*****************************************************/
        SYNC_DISK_READY:
        case MDSDR:
           Set each MURB to SYNC_DISK_READY state;
           Set # MURBs complete = 0;
           goto SYNC_DISK_IN_PROGRESS;

        /*****************************************************/
        SYNC_DISK_IN_PROGRESS:
        case MDSDP:
           if (# MURBs complete == # MURB entries)
              if (MDB "Abort Sync Request" bit == 1)
                 goto BREAK_MIRROR;
              else
                 goto WRITE_SYNC_RECORD;
           else
              if (MDB "Abort Sync Request" bit == 1)
                 "Abort Sync Request" bit = 1 in non-idle MURBs;

           call RUNMURBS;
           break;
```

```
/*****************************************************/
BREAK_MIRROR:
case MDBRM:
   Set each MURB to BREAK_MIRROR state;
   Set # MURBs complete = 0;
   call RUNMURBS;

   if (MDB "Abort Sync Request" bit == 1)
      call unpend_caller;

   goto WRITE_SYNC_RECORD;


/*****************************************************/
UNSYNC_MIRROR:
case MDUSM:
   Set each MURB to UNSYNC_MIRROR state;
   Set # MURBs complete = 0;
   call RUNMURBS;

   goto WRITE SYNC RECORD;


/*****************************************************/
DELETE_MIRROR:
case MDDLM:
   Set each MURB to DELETE MIRROR state;
   Set # MURBs complete = 0;
   call RUNMURBS;

   Set State to IDLE;
   call unpend_caller;


/*****************************************************/
WRITE_SYNC_RECORD:
case MDWSR:
   Set each MURB to WRITE_SYNC_RECORD state;
   Set # MURBs complete = 0;
   call RUNMURBS;

   Set state to IDLE;
   if ("Dont unpend caller" bit == 0)
      call unpend_caller;
   break;


/*****************************************************/
WRITE_SYNC_IN_PROGRESS_RECORD:
case MDWSR:
   Set each MURB to WRITE_SYNC_IN_PROGRESS_RECORD state;
   Set # MURBs complete = 0;
   call RUNMURBS;
```

```
            goto SYNC DISK READY;

         } /* end of switch */
      } /* end of for loop */

} /* end of MIROP.P */
```

RUNMURBS is the LDU Management service that runs Mirrored Unit Request Blocks. Each MURB runs the MDB request for the specific physical unit in the mirrored LDU. The MURB chain is traversed, and each MURB is run individually. When one MURB finishes, the number of MURBs finished is incremented in the MDB (MDB_addr->MDCMRB++), and the next MURB on the chain is run. When all MURBs have completed, the MDB request is complete. The algorithm of the sequence of events that occurs in RUNMURBS is the following:

```
RUNMURBS ( )
{
   for (each MURB on MDB_addr->MDMRB.W)
     {
     Get the MURB state and dispatch;

     switch ( MURB_addr->MUSTATE )
        {
        /*****************************************************/
        IDLE:
        case MDIDL:
           break;


        /*****************************************************/
        SYNC_DISK_READY:
        case MUSDR:
           Get Memory Buffer if None;
           Set State to SYNC_DISK_IN_PROGRESS;
           Initialize MURB info;
           Set Up MURB with initial Bitmap info;
           goto SYNC DISK IN PROGRESS;


        /*****************************************************/
        SYNC_DISK_IN_PROGRESS:
        case MUSDP:
           if (IOP bit in Buffer Header == 1)
              break;

           Calculate next transfer;
           if (none left to transfer)  or
              ("Abort Sync Request" bit == 1)
              {
              Set state to IDLE;
              Increment # MURBs Complete in MDB;
              break;
              }
```

```
   Issue Mirror Bulk Copy;
   break;


/*****************************************************/
BREAK_MIRROR:
case MUBRM:
   If (MDB "Switch" bit == 1)
      switch UDBs in Mirror UDB (if not a half mirror);

   Set Mirror State to HALF_MIRROR;
   Set State to IDLE;
   Increment # MURBs Complete in the MDB;
   break;


/*****************************************************/
DELETE_MIRROR:
case MUDLM:
   Set mirror state to DELETED;
   Release Memory buffer (used for DIB and Bitmap);
   Set State to IDLE;
   Increment # MURBs Complete in the MDB;
   break;


/*****************************************************/
SYNC_MIRROR:
case MDSYM:
   Set mirror state to SYNCHRONIZED;
   Set State to IDLE;
   Increment # MURBs Complete in the MDB;
   break;


/*****************************************************/
UNSYNC_MIRROR:
case MUUSM:
   Set mirror state to UNSYNCHRONIZED;
   Set State to IDLE;
   Increment # MURBs Complete in the MDB;
   break;


/*****************************************************/
WRITE_SYNC_RECORD:
case MUWSR:
   Get Memory Buffer if None;
   Read DIB from first UDB;
   Put in the new sync record;
   DIB "Disk is mirrored" bit = 1;
   Write first DIB back out;

   Read DIB from second UDB;
   Put in the new sync record;
   DIB "Disk is mirrored" bit = 1;
   Write second DIB back out;
```

```
            Set State to IDLE;
            Increment # MURBs Complete in the MDB;
            break;


         /*****************************************************/
         WRITE_SYNC_IN_PROGRESS_RECORD:
         case MUWSP:
            Get Memory Buffer if None;
            Read DIB from first UDB;
            Put in the new sync record;
            DIB "Disk is mirrored" bit = 1;
            Write first DIB back out;

            Read DIB from second UDB;
            Put in the new sync record;
            DIB "Disk is mirrored" bit = 1;
            DIB "Sync in progress" bit = 1;
            Write second DIB back out;

            Set State to IDLE;
            Increment # MURBs Complete in the MDB;

         } /* end of switch */
      } /* end of for loop */

} /* end of RUNMURBS */
```

LDU Management must support initialization of mirrored LDUs. However, the mirror-specific code that is actually embedded in LDU Initialization has been extracted and placed in this section. This provides a more organized and hopefully clearer format within this internals manual.

The "routine" that LDU Initialization called from XINIT.P has been designated "validate_mirror_for_LDU_init". It is called after both physical unit lists of both LDU images have been validated for correctness, but excluding mirror-related validations. Validate_mirror validates that the specified physical unit lists can indeed be initialized as one mirrored LDU. Validate_mirror creates and arranges the mirror databases as well.

For each image there exists one LCB, and for each physical unit there exists one UDB. The UDBs are currently linked to the LCB as well as the DCT. Validate_mirror will end up deallocating one of the LCBs, leaving a single LCB for the mirrored LDU. Validate_mirror will not deallocate any of the UDBs, since they must remain linked to the DCTs of their devices. However, the UDB representing each physical unit will be unlinked from the LCB. A new "Mirror UDB" will be created and linked to LCB_addr->LBUDP.W.

As a point of interest, one mirroring-related operation must be called from LDU Release. UPDMIR.P is called from DRLSE.P to write the DIB of each physical unit in the mirrored LDU. This must be done because certain words in the DIB are unique for a unit that is part of a mirrored LDU.

The C-based pseudo-code for validate_mirror_for_LDU_init is the following. The calling sequence can be found in Section 6.4.

```
validate_mirror_for_LDU_init ();
{
   /*************************************************************
    * Validate all kinds of mirror information to ensure that   *
    * these LDUs can indeed be mirrored.                        *
    *                                                           *
    * MIRINFO.P: checks that LDU unique IDs are different,      *
    *   time stamps (DIB_addr->IBDAT.W,IBTOD.W) are the same,   *
    *   and logical disk names (from DIB name space) are same.  *
    *                                                           *
    * MCONFIG.P: checks that LDU max size (LCB_addr->LBMSH.W)   *
    *   is the same, the drives to be mirrored are on the same  *
    *   controller (UDB_addr->UDDCT.W same), and that the       *
    *   controller is a unicorn (ARGUS) device that supports    *
    *   hardware mirroring.  Also, the BBTs of both units       *
    *   must be exactly equivalent!                             *
    *************************************************************/

     call MIRINFO.P;                  /* Validate mirrored LDUs */
     call MCONFIG.P;                  /* More validation checks */

   /*************************************************************
    * Allocate mirrored LDU databases.                          *
    *                                                           *
    * GETMDB.P: Find MDB on MDBCHN.W.  If it does not exist,    *
    *   create an MDB, and create and link its MURBs to the     *
    *   MDB MURB chain, MDB_addr->MDMRB.W.   Init MDB/MURBs.    *
    *                                                           *
    * GETUDB.P: Allocate a "mirror UDB" for each unit in the    *
    *   mirrored LDU.  The "primary" and "secondary" UDB        *
    *   pointers are stored in the "mirror UDB."  The mirror    *
    *   UDBs are then linked together (UDB_addr->UDUDL.W).      *
    *************************************************************/

     call GETMDB.P;                   /* Create MDB and MURBs   */
     call GETUDB.P;                   /* Create mirror UDB chain*/

   /*************************************************************
    * Link the mirror UDBs to the LCB UDB chain.                *
    *                                                           *
    * NONEED.P: Release the unneeded memory that had been       *
    *   allocated to the second image, including its LCB.       *
    *   The main "for" loop allocated an LCB for BOTH images,   *
    *   but only one is needed for the mirror LDU.  HOWEVER,    *
    *   the UDBs corresponding to each PU are not released now  *
    *   because they are linked through the DCT unit list,      *
    *   DCT_addr->DCPUL.W.  They will be released upon LDU      *
    *   RELEASE.                                                *
    *                                                           *
    * LNKMDB.P: links MDB to MDBCHN.W tail.                     *
    *************************************************************/
```

```
    LCB_addr->UDUDP.W = MDB_addr->MDMRB.W->MUUDB.W;

    call NONEED.P;                  /* Release some memory    */
    call LNKMDB.P;                  /* Link MDB to MDBCHN.W   */

   /*************************************************************
    * Synchronize the mirror.                                   *
    * SET_START: stores the mirror state (MDSYM) in the MDB     *
    *   (MDB_addr->MDSTATE), saves the data/time of sync in     *
    *   the MDB, and unpends the System Manager Task to run     *
    *   the sync request.  MUNSYNC will be stored in the DIB    *
    *   until the mirrored LDU is released.  If the MDB state   *
    *   is still MDSYM, MUNSYNC will be changed to MSYNC in     *
    *   the DIB.                                                *
    *************************************************************/

    call SET_START.P (MUNSYNC, MDSYM, MDB_addr);

} /* end of validate_mirror_for_LDU_init */
```

7 Unit Management

7.1 Overview

There are three NON-DISK unit type devices present in AOS/VS 7.50. They are:

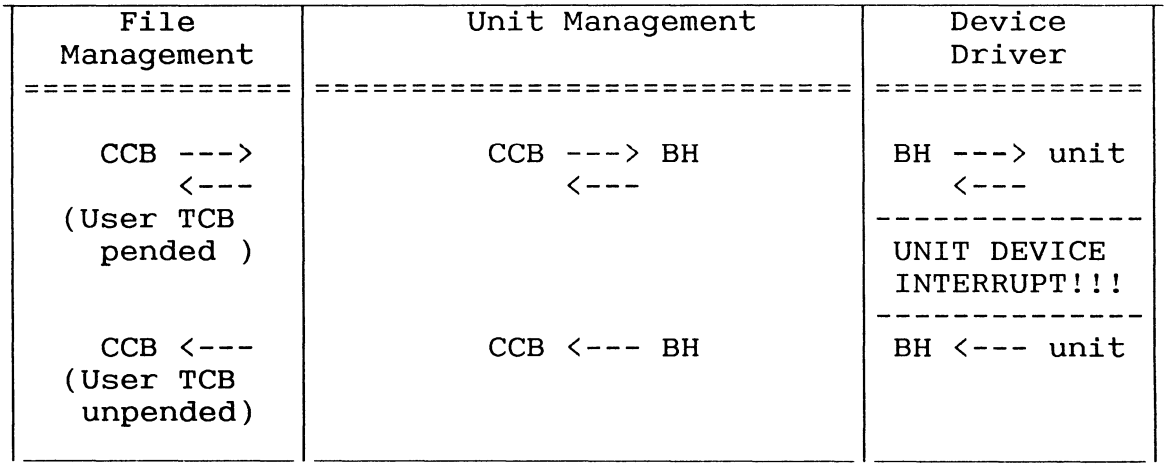
1) Magnetic Tape Units (MTUs);
2) Multiprocessor Communications Adaptor Units (MCUs); and
3) Line Printer Units (LPUs).

Generally, in this document, each unit type will be referred to by its acronym (in parentheses above). This format eliminates confusion among the various types that exist for each unit. For example, the various types of magnetic tape units (MTB, MTC, MTD, and MTJ) have been consolidated by the single acronym "MTU." The various types of line printer units (LPB, LPD, LPE, and LPJ) have been consolidated by the single acronym "LPU." Although there is only one type of multiprocessor communications adaptor (MCA), it is usually referred to as an "MCU" to preserve consistency. Note that the AOS/VS file type for MTB devices is "MTU," the file type for MCA devices is "MCU," and the file type for LPB devices is "LPU." Unicorn type devices (MTJ and LPJ) are properly distinguished when relevant. It is not an objective of this section to address physical disk unit management.

Unit Management is a file system component between File Management and the device drivers. File Management initiates logical disk CCB requests as well as unit CCB requests. Unit CCB requests are ordinary I/O requests to NON-DISK unit type devices. Direct block I/O calls (e.g., ?RDB, ?WRB) are generally used for unit I/O, although the AGENT will convert record I/O calls (e.g., ?READ/?WRITE). Request-specific information is gathered from the caller and stored in the CCB, which is "enqueued" by File Management to Unit Management. Unit Management moves the request data from the CCB to the unit I/O buffer header. Moreover, Unit Management interfaces directly with unit device driver routines to format the buffer header, enqueue the buffer header to the DCT (non-Unicorn devices) or UDB (Unicorn devices), start the device, and initiate the actual data transfer. Unlike CCB Request Management, Unit Management eliminates the intervention of Buffer Management by performing all buffer header operations independently.

In summary, Unit Management enqueues a buffer to the proper unit, initiates the data transfer by directly calling specific unit driver routines, and immediately returns to File Management. Similar to logical disk request service, File Management does not pend the calling process' control block, but returns to the system call processor, leaving the caller's TCB pended. When the unit controller interrupts the host, the data transfer will be complete. Finally, the unit buffer header post-processor unpends the requestor's TCB.

The following diagram summarizes the relationships between the components involved with Unit Management.



7.2 Unit Parameter Redefinitions

Unit devices possess attributes that disks lack, and vice versa. Nevertheless, I/O is the only means of communication with both device types. Furthermore, unit devices form part of the AOS/VS file system, and File Management databases are allocated for them. Since some essential unit device attributes, which are stored in File System databases, are non-existent for disks, parameter redefinition within these databases saves allocating unnecessary memory to store them.

The following represent the CCB parameter redefinitions for the three types of AOS/VS unit file types.

Offset		CCB Redefinitions
CBERR	0	Error code from I/O processing (all units).
CBBHR.W	7	Buffer Header address (all units).
CBMDC.W	11	DCT address (MCU).
CBBLN	13	Block length (MTU).
CBQBC	17	Requested byte count (LPU).
CBMCL	17	File number (MTU), Link number (MCU).
CBDEN	30	Density mode (MTU).

The following FCB and UDB parameters are redefined for magnetic tape units. Notice the common area in the FCB and UDB. Moreover, this common area lies within the FCB Funny FIB. When the MTU is opened, the FCB is created and initted before the UDB. The Funny FIB is read from disk into the FCB. Therefore, the common FCB offsets are copied (via WBLM instruction) into the UDB. The UDB is then updated during the time the tape unit is open. When the tape unit is closed, the driver copies the current common data from the UDB back to the FCB, which Unit Management subsequently flushes to disk. This allows Unit Management to acquire the current tape position when the tape unit is reopened. There are no FCB/UDB redefinitions for MCUs or line printers.

Offset		FCB Redefinitions
FBLFL	13	Logical EOT file number.
FBLBL	14	Logical EOT block number.
FBFIL	15	Current file number.
FBBLK	16	Current block number.
FBOBL	17	Most recent old block count.
FBOB2	20	Second most recent old block count.

Offset		UDB Redefinitions
UDWFG	17	Write flag for close.
UDLFL	20	Logical EOT file number.
UDLBL	21	Logical EOT block number.
UDFIL	22	Current file number.
UDBLK	23	Current block number.
UDOBL	24	Most recent old block count.
UDOB2	25	Second most recent old block count.
UDST1	31	Second status word on 6026 type tape drives.
UDERS	74	Erase count (after bad tape).

The buffer header is the object enqueued to the device's UDB. Request-specific data is retrieved from the buffer header and moved to the UDB by the unit's driver. Since unit request data is different from disk request data (file number, link number, etc.), there are several buffer header parameter redefinitions as well.

Offset		Buffer Header Redefinitions
BQCCB.W	0	CCB address (all units).
BQDCT.W	2	DCT address (MCU).
BQES1	4	PIO Error Status 1 (MTU).
BQOWC	4	Original Word Count (MCU).
BQORC	4	Original Byte Count (LPU).
BQES2	5	PIO Error Status 2 (MTU).
BQPRI	6	Priority (all units).
BQWDC	7	Word Count (MTU, MCU).
BQQBC	7	Requested Byte Count (LPU).
BQFIL	22	File Number (MTU).
BQLNK	22	Link Number (bits 0-7), status (bits 8-15) (MCU).
BQTMP.W	22	Temporary variable (LPU).
BQBLK	23	Block Number (MTU).
BQRTY	23	Retry Count (MCU).
BQERC	40	Error (retry) Count (MTU).
BQBYT	40	Byte flag (LPU).
BHULN	41	Length of BH for MTU/MCU/LPU

7.3 Opening Unit Files

Units are opened by the same File Management service provided by File Open Services: GOPEN.P. However, each unit type file requires different database initialization in order to make unit I/O possible. Therefore, Unit Management provides File Management with a service that takes care of the unit-specific initialization. This service is called UNIT.P.

UNIT.P is called from within GOPEN.P to open unit type files. UNIT.P is the generic entry point for all unit type files opens, and it performs ACL checks and C2 record logging. It sets the density mode flags (for MTUs) and disable form feed bit (for LPUs) in the CCB, based on the options specified by the unit opener in the ?GOPEN packet. Finally, UNIT.P calls UOPEN.P, the routine which simply initializes stack offsets and dispatches to the actual unit-specific open routine.

UOPEN.P is illustrated for each unit type file. Because the unit dispatch table labels for specific units are local to AOS/VS module UNIT.SR, the global entry point UOPEN.P is repeated for each unit open description (instead of choosing the local label as the procedure's main entry point). The contents of the algorithm are specific to the unit being described.

The inputs to UOPEN.P are the file/link number (MTUs/MCUs), the file type (index into the dispatch table), and the CCB address. The return from UOPEN.P transfers control back to UNIT.P, which, if no errors occurred, returns to GOPEN.P to continue with the opening of the file. If UOPEN.P detected an error, UNIT.P is responsible for cleaning up and returning the error code to the calling process.

7.3.1 Opening Magnetic Tape Units (MTUs)

The following C-based algorithm illustrates the MTU-specific operations that must be performed when opening the unit:

```
UOPEN.P (file_num, file_type, CCB_addr)
{
/*************************************************************
 * File_type is ?FMTU.                                       *
 * Dispatch to the Magnetic Tape Unit open routine.          *
 *************************************************************/

   LDSP (file_type, dispatch_table);

MTUNT:                          /* Magnetic Tape Unit open routine */

/*************************************************************
 * Do some initialization.                                   *
 *************************************************************/

   set_bit (CCB_addr, BCBUN);             /* Set CCB "unit" bit */
   CCB_addr->CBSTS |= CUTMT;              /* Specify MTU type   */

   if (file_num != -1)                    /* File num specified?*/
     {                                    /* Yes.               */
     CCB_addr->CBMCL = file_num;          /* Put it in the CCB, */
     set_bit (CCB_addr, BCBFN);           /* and set CCB bit to */
     }                                    /* reflect this.      */


/*************************************************************
 * Allocate a buffer header.                                 *
 * This buffer header will hold the unit-specific commands   *
 * and request parameters sent to the tape controller.  There *
 * is only one buffer header, implying that this file opener *
 * can issue only one command at a time.  Furthermore, since *
 * the unit is exclusively opened, only this process can     *
 * issue commands to the MTU while it is opened.  Two buffer *
 * headers may not be enqueued to the unit at the same time. *
 * Note:  Since only user processes do unit I/O, the data    *
 * address at BH_addr->BQADR.W will always point to a user   *
 * buffer!  THIS IS NOT A SYSTEM CACHE BUFFER!               *
 *************************************************************/

   BH_addr = GSMEM (BHULN);               /* Allocate the BH.   */
   CCB_addr->CBBHR.W = BH_addr;           /* Save addr in CCB.  */

/*************************************************************
 * Copy the density mode flags to the buffer header.         *
 * Unit Mgmt always forces MTUs to be exclusively opened.    *
 *************************************************************/

   BH_addr->BQERC = CCB_addr->CBDEN;      /* Density mode to BH.*/
   set_bit (CCB_addr->CBFCB.W, BFBEO);    /* Implicit excl open.*/
```

```
/*************************************************************
 * Allocate a UDB.                                           *
 * Save its address in both the FCB and buffer header.       *
 * (The DCT address is stored in UDB_addr->UDDCT.W)          *
 *************************************************************/

   call GTUDB.P (FCB_addr->FBDCU, &UDB_addr); /* Get/init UDB.*/
   FCB_addr->FBUDB.W = UDB_addr;              /* Save addr in */
   BH_addr->BQUDB.W  = UDB_addr;              /* FCB and BH.  */


/*************************************************************
 * Unicorn tape drives (MTJ types) can be set for two special *
 * modes (in the ?GOPEN packet): streaming mode and/or       *
 * buffered I/O mode.  If either or both of them have been   *
 * specified, set the corresponding UDB bits.  Furthermore,  *
 * unicorn controllers maintain the data stored in the FCB/UDB*
 * common area.  Thus, the data need not be copied to the UDB.*
 * This must be done, however, for non-Unicorn types.        *
 *************************************************************/

   if ( check_bit (UDB_addr, BDUCN) )   /* Unicorn device?    */
     {                                  /* Yes.               */
     if (streaming_mode)                /* STR mode specified?*/
       set_bit (UDB_addr, BUDSTR);      /* Yes, set UDB bit.  */

     if (buffered_IO_mode)              /* BIO mode specified?*/
       set_bit (UDB_addr, BUDBIO);      /* Yes, set UDB bit.  */
     }
   else
     /* Non-unicorn device. Move common area from FCB to UDB. */
     wblm (&FCB_addr->FBLFL, &UDB_addr->UDLFL, FBCOM);


/*************************************************************
 * Get the current initial status of the MTU.                *
 * MQBHR enqueues the buffer header with the "get status"    *
 * command to the tape unit.  If an error occurs on the I/O, *
 * BWAIT will take the error return, which means that the    *
 * tape drive cannot be opened because of a physical problem, *
 * e.g., "physical unit failure" or "physical unit off line." *
 * This error code is returned to the caller.                *
 * Return to GOPEN.P when done.                              *
 *************************************************************/

   BH_addr->BSTS = QTSTS;               /* Get status command.*/
   call MQBHR (&PPBSY, BH_addr);        /* Enqueue it to MTU. */
   call BWAIT;                          /* Pend until done.   */

   return;                              /* Back to GOPEN.P    */

} /* end of UOPEN.P for MTUs */
```

7.3.2 Opening Multiprocessor Communications Adaptors (MCUs)

The following C-based algorithm illustrates the MCU-specific operations that must be performed when opening the unit:

```
UOPEN.P (link_num, file_type, CCB_addr)
{
/*************************************************************
 * File_type is ?FMCU.                                       *
 * Dispatch to the MCA Unit open routine.                    *
 *************************************************************/

   LDSP (file_type, dispatch_table);

MCUNT:                                  /* MCA Unit open routine */

/*************************************************************
 * Examine link number and test its validity.                *
 * A value of -1 means that the link will be provided        *
 * dynamically with the I/O request specifications.          *
 * If the link number is valid, then set the CCB "unit" bit. *
 *************************************************************/

   CCB_addr->CBMCL = -1;                /* Assume no link num.*/
   if (link_num != -1)                  /* Link num specified?*/
     {                                  /* Yes.               */
     if (link_num < 0 || link_num > 15.)/* Is it legal?       */
       return (Illegal_link_number);    /* No, return error.  */

     CCB_addr->CBMCL = file_num;        /* Put it in the CCB. */
     set_bit (CCB_addr, BCBFN);         /* and set permanent  */
     }                                  /* link for channel.  */

                                        /* SHOW THIS IS UNIT! */
   set_bit (CCB_addr, BCBUN);           /* Set CCB "unit" bit */
   CCB_addr->CBSTS |= CUTMT;            /* Specify MCU type   */


/*************************************************************
 * Allocate a buffer header.                                 *
 * This buffer header will hold the unit-specific commands   *
 * and request parameters sent to the MCA controller.  There *
 * is only one buffer header, implying one channel request at *
 * a time.  The MCU is not opened exclusively, so there can  *
 * actually be multiple commands simultaneously enqueued to  *
 * the unit.                                                 *
 *                                                           *
 * Since only user processes do unit I/O, the data address at *
 * BH_addr->BQADR.W will always point to a user buffer!      *
 * THIS IS NOT A SYSTEM CACHE BUFFER!                        *
 *************************************************************/
```

```
    BH_addr = GSMEM (BHULN);            /* Allocate the BH.     */

    CCB_addr->CBBHR.W = BH_addr;        /* Save BH addr in CCB. */
    BH_addr->BQCCB.W  = CCB_addr;       /* Save CCB addr in BH. */

    DCT_addr = &BTBL + (2 * CCB_addr->CBFCB.W->FBDCU);
    BH_addr->BQDCT.W = DCT_addr;        /* Save DCT addr in BH. */
    CCB_addr->CBMDC.W = DCT_addr;       /* Save DCT addr in CCB.*/

    return;                             /* Back to GOPEN.P      */

} /* end of UOPEN.P for MCUs */
```

7.3.3 Opening Line Printer Units (LPUs)

The following C-based algorithm illustrates the LPU-specific operations that must be performed when opening the unit:

```
UOPEN.P (not_used, file_type, CCB_addr)
{
/*************************************************************
 * File_type is ?FLPU, ?FLPD, or ?LPE.                       *
 * Dispatch to the correct LPU unit open routine.            *
 *************************************************************/

   LDSP (file_type, dispatch_table);

LPUNT:                          /* Line Printer Unit open routines.*/

 *************************************************************
 * Store the correct unit type in the CCB.                   *
 *************************************************************/

   switch (FCB_addr->FBTYP)             /* What type LP is this?*/
     {
     case ?FLPU:                        /* ?FLPU ?              */
       CCB_addr->CBSTS |= CULPB;        /* Yes, LPB type to CCB */
       break;

     case ?FLPD:                        /* ?FLPD ?              */
       CCB_addr->CBSTS |= CULPD;        /* Yes, LPD type to CCB */
       break;

     case ?FLPE:                        /* ?FLPE ?              */
       CCB_addr->CBSTS |= CULPE;        /* Yes, LPE type to CCB */
       break;
     }

/*************************************************************
 * Set the CCB "unit" bit.                                   *
 *************************************************************/

   set_bit (CCB_addr, BCBUN);             /* Set CCB "unit" bit */

/*************************************************************
 * Allocate a buffer header.                                 *
 * This buffer header will hold the unit-specific commands   *
 * and request parameters sent to the line printer.          *
 * Then do some other database initializations.              *
 *************************************************************/

   BH_addr = GSMEM (BHULN);             /* Allocate the BH.     */

   CCB_addr->CBBHR.W = BH_addr;         /* Save BH addr in CCB. */
   BH_addr->BQCCB.W  = CCB_addr;        /* Save CCB addr in BH. */
```

```
   DCT_addr = &BTBL + (2 * CCB_addr->CBFCB.W->FBDCU);  /* DCT */
   BH_addr->BQDCT.W = DCT_addr;        /* Save DCT addr in BH. */

   set_bit (CCB_addr->CBFCB.W, BFBEO);  /* Implicit excl open */
   if ( check_bit (CCB_addr, BCBFF) )   /* "Disable FF" bit   */
     set_bit (DCT_addr, BDDFF);         /* from CCB to DCT.   */


/*************************************************************
 * If the line printer is on a Unicorn controller, the       *
 * controller will need to access a UDB, which will be       *
 * allocated now.  Non-Unicorn devices do not require a UDB. *
 * This is because buffer headers are enqueued off the UDB   *
 * request queue (UDB_addr->UDCRQ.W) for Unicorn devices, but *
 * off the DCT request queue (DCT_addr->DCCRQ.W) for         *
 * non-Unicorn devices.                                      *
 * Additionally for Unicorn devices, we must get the device's *
 * initial status and save it in the buffer header.          *
 *************************************************************/

   if (check_bit (UDB_addr, BDUCN) )    /* Unicorn device?    */
     {                                  /* Yes.               */
     call GTUDB.P (FCB_addr->FBDCU,&UDB_addr);/* Get/init UDB.*/

     FCB_addr->FBUDB.W = UDB_addr;      /* Save addr in       */
     BH_addr->BQUDB.W  = UDB_addr;      /* FCB and BH.        */

     BH_addr->BSTS = QTSTS;             /* Get status command.*/
     call MQBHR (&PPBSY, BH_addr);      /* Enqueue it to LPU. */
     call BWAIT;                        /* Pend.  I/O errors  */
                                        /* returned to caller.*/

     call *DCT_addr->DCTFR.W;           /* Format tab memory  */
     return;                            /* and return to      */
     }                                  /* GOPEN.P.           */

   else
     {                                  /* Not unicorn device!*/
     call *DCT_addr->DCTFR.W;           /* Format tab memory  */
     return;                            /* and return to      */
     }                                  /* GOPEN.P.           */

} /* end of UOPEN.P for LPUs */
```

7.3.4 Opening Disk Units (DKUs)

LDU Management handles the initialization ("open") and release ("close") of logical disk units. Unit Management is responsible for providing File Open/Close Services with routines to allocate unit-specific databases for disk units when they are opened for physical I/O. Although physical disk unit opens and closes have been incorporated into Unit Management, the discussion on physical disk I/O is covered by Buffer Management (section 5.13).

The following C-based algorithm illustrates the DKU-specific operations that must be performed when opening the unit:

```
UOPEN.P (file_num, file_type, CCB_addr)
{
/*************************************************************
 * File_type is ?FDKU.                                       *
 * Dispatch to the Disk Unit open routine.                   *
 *************************************************************/

   LDSP (file_type, dispatch_table);

DUNT:                          /* Physical Disk Unit open routine */

/*************************************************************
 * If the unit is already open, all necessary databases must *
 * exist; there is nothing to do except return.              *
 * The "unit" bit (BCBUN) is not set in the CCB!  Physical   *
 * disk I/O is performed quite differently from other unit   *
 * I/O.  This is why the BCBUN bit is not set AND why physical*
 * disk I/O is explained in Buffer Management.               *
 *************************************************************/

   FCB_addr = CCB_addr->CBFCB.W;          /* Save FCB addr      */
   if (FCB_addr->FBOPN >= 1)              /* Is unit open?      */
     return;                              /* Yes, go home.      */


/*************************************************************
 * Allocate an LCB.                                          *
 * An LCB is allocated for a single unit for the sole purpose *
 * of maintaining the UDB pointer.  When Physical Disk I/O   *
 * Services calls Buffer Management service NQLDRQ to enqueue *
 * a buffer header to the disk unit, it must pass an LCB.    *
 *************************************************************/

   LCB_addr = GSMEM (LBBLT);              /* Allocate the LCB.  */
   FCB_addr->FBLCB.W = LCB_addr;          /* Save addr in FCB.  */

/*************************************************************
 * Allocate and init a UDB.                                  *
 * GDUDB.P calls the device-specific UDB init routine, which *
 * inits such UDB parameters as disk size, cylinders, etc.   *
 * It also links the UDB onto the DCT physical unit list.    *
 *************************************************************/

   call GDUDB.P (FCB_addr->FBDCU, &UDB_addr);
   LCB_addr->LBUDB.W = UDB_addr;          /* UDB addr to LCB.   */

/*************************************************************
 * Finish initializing the UDB, FCB and LCB.                 *
 * Set the first/last unit addresses. (GDUDB.P set UDLAH.W!)  *
 * Disk's elementsize is equal to block size of whole disk.  *
 * The EOF is equal to the number of blocks * 512.           *
 *************************************************************/

   UDB_addr->UDFAH.W = 1;                 /* First addr = 1 (UDB).*/
   FCB_addr->UDFAH.W = 1;                 /* First addr = 1 (FCB).*/
   UDB_addr->UDLAH.W += 1;                /* Size = size + 1.     */
```

```
   FCB_addr->FBDFH.W = UDB_addr->UDLAH.W; /* Elementsize, EOF.*/
   FCB_addr->FBEFH.W = UDB_addr->UDLAH.W * BYTES_PER_BLOCK;

   FCB_addr->FBIDX = 0;                   /* Max index levels = 0 */
                                          /* makes file contiguous*/

/*************************************************************
 * Link LCB to global LCB chain (tail).                      *
 * For the sake of consistency, show LCB cache buffer list   *
 * queue descriptor as empty.  No buffer headers will ever   *
 * be enqueued to the LCB because ONLY physical I/O is done. *
 *************************************************************/

   LCB_addr->LBCLP.W = -1;                /* Make clean, null     */
   LCB_addr->LBCLB.W = -1;                /* LCB queue descriptor.*/

   LCB_addr->LBLBP.W = -1;                /* Last LCB in chain.   */
   *.ELCB.W->LBLBP.W = LCB_addr;          /* Link LCB to tail.    */
   .ELCB.W = LCB_addr;                    /* New LCB tail pointer.*/

   return;

} /* end of UOPEN.P for DKUs */
```

7.4 Closing Unit Files

Units are closed by the same File Management service provided by File Close Services: GCLOSE.P. Just as Unit Management provides File Open Services with the unit-specific open routine UOPEN.P, so does it provide File Close Services with the converse routine to perform unit-specific close routine. This service is called UCLOSE.P.

UCLOSE.P essentially deallocates the file system databases allocated by UOPEN.P. The service is short and straightforward. The algorithm that describes UCLOSE.P is illustrated below.

The inputs to UCLOSE.P are the file type and the FCB address. The return (WRTN) from UCLOSE.P transfers control back to GCLOSE.P to continue with the normal closing of the file.

```
UCLOSE.P (file_type, FCB_addr)
{
/************************************************************
 * If UOPEN.P allocated a UDB for the unit, deallocate it.    *
 * If UOPEN.P allocated an LCB for the unit, deallocate it.   *
 * THE BHS THAT UOPEN.P ALLOCATED ARE DEALLOCATED BY GCLOSE.P!*
 * In some cases, call a routine or send a command to the     *
 * unit controller to "reinitialize" (get it ready) for the   *
 * next time it is opened.                                    *
 ************************************************************/

   switch ( file_type )
     {
     case ?FMTU:

     /*******************************************************
      * Call mag tape cleanup routine (in the DCT).         *
      * Release the UDB's memory.                           *
      *******************************************************/

       call *FCB_addr->FBUDB.W->UDDCT.W->DCCLN.W;
       call RLUDB.P (UDB_addr);
       break;

     case ?FMCA:

     /*******************************************************
      * Remember, BH deallocated by GCLOSE.P, and MCUs do   *
      * not require UDBs, so there is nothing to do!        *
      *******************************************************/

       break;

     case ?FLPU:
     case ?FLPD:
     case ?FLPE:

     /*******************************************************
      * Release UDB for unicorn line printers.              *
      *******************************************************/
```

```
        if ( check_bit (DCT_addr, BDUCN) )
          call RLUDB.P (UDB_addr);
        break;

      case ?FDKU:

      /*****************************************************
       * Call disk unit recalibration routine, which sets   *
       * the disk heads to cylinder 0, track 0, sector 0.   *
       * Release the UDB.                                   *
       * Unlink LCB from global LCB chain release it.       *
       *****************************************************/

        call RECAL (FCB_addr->FBLCB.W->LBUDP.W);
        call RLUDB.P (UDB_addr);

        unlink (&.MLCB.W, LCB_addr);       /* Unlink LCB,     */
        call RSMEM (LCB_addr, LBBLT);      /* and release it.*/

        break;
      }

} /* end of UCLOSE.P */
```

7.5 Unit I/O Interface Services

Since unit devices are represented by AOS/VS filenames, they are not managed exclusively by Unit Management. For example, generic File Open/Close Services (e.g., GOPEN.P, GCLOSE.P) handles most of the general open/close work, but calls to Unit Management UOPEN.P/ UCLOSE.P routines complete the open/close of the unit. Similarly, File Management I/O Interface Services (e.g., RDB.P, WRB.P) accomplishes the task of extracting the unit I/O request parameters from the user packet and initializing the user CCB. File Management then enqueues the unit I/O to the device by calling the Unit Management I/O Enqueue routine UIOENQ.

The control block that Process Management allocates to run any Unit Management I/O system call does not pend after the unit I/O is enqueued. Since the call to UIOENQ is the very last operation performed by these services, they are able to avoid invoking the CB pending mechanism after enqueuing the request because there would be nothing to do, except return, when the request completed. Hence, just like RDB.P/WRB.P for logical disk I/O (Section 3.4.5), the unit I/O system call interface enqueues the request, zeroes out the TCB address in the control block, and immediately returns. The system call processor recognizes that CB_addr->CATCB.W is zero, and thus releases the control block, but keeps the TCB pended. The unit post-processor unpends the TCB, whose address is stored at offset CBTCB.W of the CCB, when the I/O request is complete.

The unit I/O system call interface services are presented in the C-based algorithm below. These entry points are the same ones used for logical and physical disk I/O. Due to the identical format of the packets, common entry points were established for purposes of initial validation and data manipulation. At assembly language level, the code makes checks to determine what type of service is being requested (e.g., unit I/O, physical disk I/O, ?BLKIO) and branches to the appropriate path. This algorithm follows only the unit I/O paths.

The calling process' input accumulators are represented by parameters in the TCB. The inputs and outputs to RDB.P/WRB.P are the following:

Variable	Input	Output
TAC0.W	Not used.	Undefined.
TAC1.W	Channel number.	Number of bytes actually transferred.
TAC2.W	Address of packet.	Unchanged.

```
#define SCMCG 0372              /* Unit I/O system call charge */

RDB.P/WRB.P/PRDB.P/PWRB.P/BLKIO.P  (TAC1.W, TAC2.W)
{
/*************************************************************
 * This is a system call with a packet.                      *
 * Get the necessary data into system space.                 *
 * Note: a trap handler address is stored in CCB_addr->CBFEH.W*
 * in case the wblm get an access violation.                 *
 *************************************************************/

   channel_num = TAC1.W;
   caller_pkt  = TAC2.W
   wblm (caller_pkt, &sys_pkt, pkt_len);

/*************************************************************
 * Find the (user) CCB for the input channel.                *
 * DFAULT returns error code in CERWD of the control block,  *
 * which the system call processor will interpret.           *
 *************************************************************\

   PTBL_addr = *CC.W->CPTAD.W;
   call DFAULT (channel_num, PTBL_addr, &CCB_addr);
      if (error) return;              /* Error code in CERWD */

/*************************************************************
 * Got the CCB.                                              *
 * This is where the common code for logical/physical/unit   *
 * branches off for unit devices.  The rest of this algorithm *
 * includes unit-specific code ONLY!                         *
 * Again, physical disk I/O is NOT handled by Unit Management!*
 *************************************************************/

   if ( check_bit (CCB_addr->CBSTS,CBUNTB) )  /* UNIT DEVICE? */
      {                                       /* Y E S ! ! !  */

    /*********************************************************
     * Initialize the specific unit file's CCB with the      *
     * request parameters selected by the user.              *
     *********************************************************/

      switch ( CCB_addr->CBSTS & UNIT_TYPE_MASK )
         {
         case MTU:

          /*****************************************************
           * MAGNETIC TAPE UNIT REQUEST.                       *
           * Move starting block num, block size, and          *
           * and number of blocks to transfer into the CCB.    *
           * If file num was not specified at ?GOPEN, get it   *
           * from the packet.  Then set CCB option bits if     *
           * if requested (BCBEO for ?ENOV, BCSAF for ?SAFM).  *
           *****************************************************/

            CCB_addr->CBDBL = sys_pkt->?PRNL;      /* Blk num */
            CCB_addr->CBBLN = sys_pkt->?PRCL;      /* Blk siz */
            CCB_addr->CBNBK = sys_pkt->?PSTI & 377;/* Num blks*/

            if ( !check_bit (CCB_addr->CBSTS, CBFNSB) )
               CCB_addr->CBDBH = sys_pkt->?PRNH;
```

```
 call set_special_options_in_CCB;

/*****************************************************
 * If physical I/O request (?PRDB/?PWRB/?BLKIO),     *
 * do the following: 1) set the PIO bit in the CCB.  *
 * This tells the BH post-processor to move error/   *
 * status information into the PIO packet.           *
 * 2) pin the PIO error packet pages, so the PP will*
 * not fault when transferring PIO data, 3) zero     *
 * out the PIO error packet explicitly.              *
 *****************************************************/

 if (physical I/O)
    {
    set_bit (CCB_addr, BCBPIO);        /* PIO bit=1 */

    saved_user_data_addr = CCB_addr->CBUAD.W;
    CCB_addr->CBUAD.W = error_packet_addr;
    call DMTST (PIBLT*2, CCB_addr);    /* Pin pages */
    CCB_addr->CBUAD.W = saved_user_data_addr;

    error_packet_addr->PCS1 = 0;       /* Zero out  */
    error_packet_addr->PCS3 = 0;       /* PIO pkt.  */
    error_packet_addr->PCS5 = 0;
    error_packet_addr->PCS7 = 0;
    }

/*****************************************************
 * DMTST:  Pin the user pages referenced.  This      *
 * ensures that pages will be resident during the    *
 * data transfer.  If the pages are faulted out,     *
 * the controller could corrupt a good page at the   *
 * memory address.                                   *
 * CHRG:  Charge for I/O, check time slice left.     *
 *****************************************************/

 call DMTST (CCB_addr->CBNBK * CCB_addr->CCBLN,
             CCB_addr);
 call CHRG (SCMCG, CCB_addr->CBNBK, CCB_addr);

/*****************************************************
 * UIOENQ: Enqueue the unit I/O.                     *
 * Then, since there is no more to do here, advise   *
 * the system call processor to deallocate (or       *
 * reassign) the CB.  The MTU post-processor will    *
 * unpend the user TCB, unpin the user buffer pages,*
 * and return data to the caller's packet.           *
 *****************************************************/

 call UIOENQ (CCB_addr);       /* Unit I/O ENQueue  */

 *CC.W->CATCB.W = 0;            /* Deallocate CB, but*/
                               /* keep TCB pended.  */
 return;                       /* System call done. */
```

```
case MCU:

 /*****************************************************
  * MCA UNIT REQUEST.                                 *
  * Initialize CCB with request parameters:           *
  * Move retry count, block size, link number to CCB.*
  * Since the MCA transfers only 1 block of size     *
  * ?PRCL (i.e., transfer of ?PRCL bytes), CBNBK is  *
  * used to specify the transmission mode instead of *
  * the number of blocks to transfer.                 *
  * (1 = protocol mode, 2 = direct mode)              *
  *****************************************************/

   CCB_addr->CBDBL = sys_pkt->?PRNL;  /* Retry count */
   CCB_addr->CBBLN = sys_pkt->?PRCL;  /* Block size  */

   if ( !check_bit (CCB_addr->CBSTS, CBFNSB) )
      CCB_addr->CBDBH = sys_pkt->?PRNH;  /* Link num */

   if ( extract_mode (sys_pkt->?PSTI) == direct )
      CCB_addr->CBNBK = 2;
   else
      CCB_addr->CBNBK = 1;


 /*****************************************************
  * DMTST:  Fault and pin the page in user packet on *
  * which the link number falls.  This must be done  *
  * because the MCA post-processor updates the link  *
  * number (at interrupt level).  This page must be  *
  * resident.  Then, as usual for user data          *
  * transfers, pin the user buffer pages.            *
  * CHRG: Charge for the system call.                *
  *****************************************************/

/* Link num falls on page containing ?PRNH.  Pin it! */
   call FLTPIN (&caller_pkt->?PRNH, *CMAP.W);

   call DMTST (CCB_addr->CCBLN, CCB_addr);
   call CHRG (SCMCG, 1, CCB_addr);

 /*****************************************************
  * UIOENQ: Enqueue the unit I/O.                     *
  * As usual, tell the system call processor to      *
  * deallocate the CB but to keep the TCB pended.    *
  * The TCB will later by unpended by the            *
  * post-processor.                                   *
  *****************************************************/

   call UIOENQ (CCB_addr);       /* Enqueue the I/O. */

   *CC.W->CATCB.W = 0;
   return;
```

```
      case LPU:

       /*****************************************************
        * LINE PRINTER UNIT REQUEST.                        *
        * Since users can issue only ?WRBs to a LPU, return*
        * an error if a read is specified.                  *
        * Null requests to LPUs are illegal as well.        *
        * Set CCB option bits if requested (BCBEO = ?ENOV).*
        *****************************************************/

         if (TCB_addr->TSYS.W == ?RDB)          /* Read req? */
            return (File_read_error);           /* Yes, bad. */

         if (sys_pkt->?PRCL == 0)               /* Null req? */
           return (Invalid_system_call_parameter); /* Bad. */

         call set_special_options_in_CCB;


       /*****************************************************
        * Move the number of bytes requested to the CCB.    *
        * Then, pin the user buffer pages, charge for the   *
        * system call, enqueue the I/O, tell syscall        *
        * processor to deallocate the CB but keep the TCB   *
        * pended, and return.  The LPU post-processor       *
        * PPMUS (same for MTUs) will unpend the caller's    *
        * TCB and unpin user pages when the I/O completes.  *
        *****************************************************/

         CCB_addr->CBQBC = sys_pkt->?PRCL;          /* Move   */

         call DMTST (CCB_addr->CBQBC, CCB_addr); /* Pin    */
         call CHRG (SCMCG, 1, CCB_addr);         /* Charge */

         call UIOENQ (CCB_addr);                 /* Enqueue*/

         *CC.W->CATCB.W = 0;                     /* Fix CB */
         return;                                 /* Done!  */
      }

   } /* end of if unit type */

} /* end of RDB.P/WRB.P/PRDB.P/PRWB.P/BLKIO.P
```

7.6 Enqueuing Unit I/O (UIOENQ)

Unit I/O Interface Services initializes the requesting CCB with the request parameters and calls Unit Management to "enqueue the unit I/O" (or "enqueue the unit CCB request") to the appropriate unit. Each different unit type has a unique routine to extract request data from the unit CCB, store it in the buffer header, and enqueue the buffer header to the unit device's DCT (or UDB if unicorn device) request queue. The Unit Management interface to enqueue unit I/O is UIOENQ.

UIOENQ simply takes the CCB address as an input, isolates the unit type from the status word, and calls the appropriate Unit Management operation to enqueue the CCB request to the unit. The three Unit Management routines that actually do most of the work in enqueuing the unit CCB requests are:

1) MQCCB - Enqueue a unit CCB request to a MTU;
2) MCACB - Enqueue a unit CCB request to a MCU;
3) LPBIO - Enqueue a unit CCB request to a LPU.

In a multiprocessor environment, programmed I/O to any system device must be issued by the mother job processor (JP). Before UIOENQ dispatches to the unit-specific routine to enqueue the request, it must check if it is running on the mother JP. If so, UIOENQ proceeds with the enqueue.

If the CB executing in UIOENQ is running on a daughter processor, it must signal the mother processor to enqueue the request. Since the Disk Manager Task always runs on the mother processor in AOS/VS 7.50, it takes responsibility for enqueuing unit CCB requests originating on daughter processors. UIOENQ enqueues the CCB to the global unit I/O wait queue UIOQUE.W, increments the UIOQUE.W element count (UIOQCT), calls DWAKE to unpend the Disk Manager Task, and returns. CCBs are linked to UIOQUE.W through CCB_addr-CBQLK.W. When the Disk Manager task is scheduled (on the mother processor) and determines that there is at least one CCB enqueued to UIOQUE.W, it calls UIODEQ to dequeue one unit CCB request enqueued by a daughter processor and successfully enqueue it to the appropriate device. (The Disk Manager Task is called separately for each CCB; it does only one pre-processing request at a time.) This procedure is outlined below:

Daughter JP --->

```
RDB.P ()
/***********************************************
 * Unit I/O Interface services calls UIOENQ to  *
 * unit CCB requests.                           *
 ***********************************************/
{
                :
   call UIOENQ (CCB_addr);       /* Enqueue I/O */
   CB_addr->CATCB.W = 0;
   return;
}


UIOENQ (*CCB_addr)
{
                :
   if ( check_bit (*MYPPCB.W, BCPMST) )

   /*********************************************
    * If mother JP, enqueue I/O NOW to unit.    *
    * Depending on the unit type, QUE_IT        *
    * dispatches to MQCCB, MCACB, or LPBIO.     *
    *********************************************/
      {
      call QUE_IT;
      return;
      }
   else

   /*********************************************
    * If not mother JP, enqueue CCB to the unit *
    * I/O wait queue UIOQUE.W.  The Disk Manager*
    * Task (runs on mother) will remove CCBs on *
    * UIOQUE.W and successfully enqueue the I/O.*
    *********************************************/
      {
      call enqueue_tail (&UIOQUE.W, CCB_addr->CBQLK.W);

      UIOQCT += 1;       /* Incr UIOQUE.W CCB count  */
      call DWAKE;        /* Unpend Disk Manager Task */
      return;
      }
}
```

Mother JP --->

```
RUNLC1: /* Disk Manager Task */
{
                :
   if (UIOQCT >= 1)      /* Anything on UIOQUE.W? */

   /*********************************************
    * The Disk Manager Task enqueues the unit   *
    * CCB requests to the appropriate unit.     *
    * UIODEQ: dequeues CCBs from UIOQUE.W and   *
    *         calls QUE_IT to enqueue the CCB   *
    *         to the unit's DCT request queue.  *
    *********************************************/
      {
      call UIODEQ;       /* Yes, dequeue CCB from*/
      }                  /* UIOQUE.W, and enque  */
               :         /* BH to the unit.      */
}


UIODEQ( )
{
   /*********************************************
    * If UIOQUE.W is not empty, dispatch to the *
    * appropriate routine to enqueue the I/O    *
    * to the proper device.                     *
    * Note: only one CCB processed at a time.   *
    *********************************************/
      {
      if (*UIOQUE.W != -1)          /* Any CCBs?     */
        {                           /* Yes.          */
        CCB_addr = *UIOQUE.W;       /* Get req CCB.  */
        *UIOQUE.W = CCB_addr->CBQLK.W; /* New head */
        UIOQCT -= 1;                /* One less now. */

        call QUE_IT (CCB_addr);     /* Dispatch to   */
        }                           /* enque I/O to  */
                                    /* proper routine*/
      return;
      }


QUE_IT (*CCB_addr)
{
   /*********************************************
    * This routine simply dispatches on the     *
    * unit type in the CCB to the appropriate   *
    * unit I/O enqueue routine.  The buffer     *
    * header at CCB_addr->CBBHR.W will be       *
    * enqueued to the unit.                     *
    *********************************************/

   LDSP (CCB_addr->CBSTS & UNIT_TYPE_MASK, dsp_table);
   return;
}
```

7.7 Enqueuing Unit CCB Requests to Magnetic Tape Units (MQCCB)

Before the actual buffer header enqueue is done, the buffer header must be initialized with the request data contained in the unit CCB. The Unit Management service MQCCB extracts this data from the CCB and moves it into the buffer header. MQCCB then readies the buffer header for I/O by setting the I/O-in-progress bit. Finally, MQCCB enqueues the buffer header to the appropriate place.

In order for the unit device to begin transferring data, the buffer header must be enqueued to:

a) the device's DCT request queue, DCT_addr->DCCRQ.W, if a non-unicorn device, or
b) the device's UDB request queue, UDB_addr->UDCRQ.W, if a unicorn device.

For non-Unicorn tape drives, if there are no requests on the queue, the device must be started by the magnetic tape driver start-up routine, whose address is found at DCT_addr->DCSTR.W. The start-up routine (which is a component of the driver world) initializes the UDB, processes the specific unit command, sets up the data channel map for the transfer, and issues the I/O command to the device controller. If the device is busy (rewinding) when the start-up routine is called, the input buffer header is temporarily enqueued to UDB_addr->UDCRQ.W. When the rewind completes, the start-up routine will be called again to send out the I/O command to the device. The non-Unicorn start-up routine is called MTAST.

For Unicorn tape drives, the buffer header is first enqueued to UDB_addr->UDCRQ.W by the Unicorn device buffer header enqueue routine UCNNQ. The Unicorn device start-up routine, UCNST, allocates a database called a Unicorn Control Block (UCB). The UCB contains pointers to the requesting buffer header and to the unit's UDB. UCNST enqueues the UCB to the DCT UCB list, DCUCB.W, where the Unicorn controller looks to find the base of the I/O request list. In effect, the UCB is the database that ultimately contains the request-specific information for I/O requests to Unicorn devices. Since this section does not cover device drivers in detail, there is only limited discussion on this topic.

After the buffer header is enqueued, MQCCB returns to its calling routine, UIOENQ. Consequently, UIOENQ returns immediately to its caller, Unit I/O Interface Services. Since there is no more work to do by the system call interface routine, the call will return to the system call processor (in lieu of calling CWAIT to pend), and only the requesting TCB of the caller's process will remain pended.

The C-based algorithm that clearly illustrates the MQCCB procedure follows. The only inputs are the unit-specific post-processor address and the unit CCB address.

```
#define BYTES_PER_PAGE 04000      /* Number of bytes per page */
#define SCRTY 015                 /* Default tape retry count */

MQCCB (PP_addr, CCB_addr)
{
/************************************************************
 * This unit's one and only BH was allocated when the unit  *
 * was opened.  Its address was stored in the CCB.          *
 * Save its address locally, and store the CCB addr in the BH.*
 ************************************************************/

   BH_addr = CCB_addr->CBBHR.W;           /* Save BH addr      */
   BH_addr->BQCCB.W = CCB_addr;           /* Save CCB in BH    */

/************************************************************
 * Move the unit CCB request parameters into the BH.        *
 ************************************************************/

   BH_addr->BQFIL = CCB_addr->CBDBH;      /* File number       */
   BH_addr->BQBLK = CCB_addr->CBDBL;      /* Start block num   */
   BH_addr->BQPRI = CCB_addr->CBPRI;      /* Process PNQF      */
   BH_addr->BQNBK = CCB_addr->CBNBK;      /* Num blks to xfer  */
   BH_addr->BQWDC = - CCB_addr->CBBLN;    /* Block size (words)*/

   BH_addr->BQMAP.W = CCB_addr->CBPTA.W; /* PTBL address      */
   BH_addr->BQUAD.W = CCB_addr->CBUAD.W; /* User buffer addr  */
   BH_addr->BQRBC.W = 0;                  /* Init run byte cnt */

/************************************************************
 * Init the BH some more before enqueuing it.               *
 ************************************************************/

   BH_addr->BQST = 0;                     /* Init status (read)*/
   if ( check_bit (CCB_addr, BCBC1) )     /* Is this a write?  */
      set_bit (BH_addr, BQMOD);           /* Yes, set mod bit. */

   if ( check_bit (CCB_addr, BCBPIO) )    /* Physical I/O?     */
      set_bit (BH_addr, BQPIO);           /* Yes, set PIO bit. */

   if ( check_bit (CCB_addr, BCBEO) )     /* Log EOT override? */
      set_bit (BH_addr, BQEOV);           /* Yes, set in BH.   */

   if ( check_bit (CCB_addr, BCSAF) )     /* ?SAFM option set? */
      set_bit (BH_addr, BQSAF);           /* Yes, set in BH.   */

/************************************************************
 * Calculate the maximum possible data transfer size in bytes.*
 * This is done by multiplying the number of bytes per page by*
 * the number of map slots assigned to the MTU.  (Must      *
 * subtract 1 because DCT_addr->DCNMS actually indicates one  *
 * map slot too many, due to possible page overlap of buffer.)*
 * Verify that the data transfer size is legal.             *
 ************************************************************/
```

```
   max_possible_bytes = (DCT_addr->DCNMS - 1) * BYTES_PER_PAGE;

   if ( (CCB_addr->CBBLN > max_possible_bytes) || /*  Legal   */
        (CCB_addr->CBBLN < 3) )                   /*  xfer?   */
      {                                          /*    NO!   */

      if (BH_addr->BQNBK != 0)            /* Null request?     */
         {                                /* No, return illegal*/
         return (Illegal_block_size);     /* block size error. */
         }
      }

/*************************************************************
 * MQBHR: Magnetic tape unit enQueue Buffer HeadeR.          *
 * MQCCB calls MQBHR, which has been EXPANDED here!          *
 *************************************************************/

MQBHR:

/*************************************************************
 * Enqueue the buffer header to the unit device.             *
 * If this unit is a Unicorn device, a special routine must  *
 * handle the buffer header enqueue.  While non-Unicorn      *
 * device BHs are enqueued to the DCT, Unicorn device BHs    *
 * are initially enqueued to the UDB request queue,          *
 * UDB_addr->UDCRQ.W.  Once the BH is enqueued, we can return *
 * to our caller.                                            *
 * The post-processor will unpend the requestor's TCB.       *
 *************************************************************/

   if ( check_bit (DCT_addr, BDUCN)      /* Unicorn device?   */
      {                                  /* Yes.              */
      call NQUCN (unpend_addr, BH_addr); /* Enqueue BH to UDB.*/
      return;                            /* Nothing else!     */
      }

/*************************************************************
 * Not a Unicorn device.                                     *
 * First finish up initializing the buffer header with the   *
 * request specifications.                                   *
 * Then, enqueue the buffer header to DCT request queue.     *
 * Buffer headers are enqueued to DCT_addr->DCCRQ.W in order *
 * of priority enqueue factor (PNQR), saved at BH_addr->BQPRI.*
 *************************************************************/

   BH_addr->BQUPD.W = unpend_addr;       /* Save BH pp addr.  */

   set_bit (BH_addr, BQTIO);             /* Set I/O-in-prog.  */
   set_bit (BH_addr, BQNAB);             /* NOT a system BH.  */

   if ( check_bit (BH_addr, BQPIO) )     /* Physical I/O?     */
      BH_addr->BQERC |= 1;               /* Yes, one retry.   */
   else                                  /* No.               */
      BH_addr->BQERC |= SCRTY;           /* Max retries.      */

   call MASK (DCT_addr->DCMSK);          /* Mask out MTU ints */
   set_bit (DCT_addr, BDLOK);            /* and get DCT lock. */
```

```
/*************************************************************
 * Enqueue the buffer header to the DCT.                     *
 * If it is the only one, call the device start-up routine.  *
 * Otherwise, the device must already be started.            *
 * Note: there can only be one BH per unit (@MTB0), but      *
 * multiple BHs per device (@MTB units 0,1,2,3).             *
 *************************************************************/

   call enqueue_by_PNQF (DCT_addr->DCCRQ.W, BH_addr, BQQLK.W);

   if (first_request_on_DCT_queue)       /* First on queue?   */
      call *DCT_addr->DCSTR.W;           /* Yes, start device.*/

   clear_bit (DCT_addr, BDLOK);          /* Release DCT lock. */
   call UNMASK;                          /* Allow MTU ints.   */

   return;                               /* Back to caller.   */

} /* end of MQCCB */
```

7.8 Enqueuing Unit CCB Requests to MCA Units (MCACB)

The Unit Management service MCACB extracts this data from the CCB and moves it into the buffer header for MCUs. MCACB then readies the buffer header for I/O by setting the I/O-in-progress bit. Finally, MCACB enqueues the buffer header to the MCA Link Table. MCUs are not Unicorn devices; the MCA driver is used to issue I/O commands to the MCU.

The MCA Link Table is used to hold data relevant to the unit's current state. Its address is stored in the DCT at DCT_addr->DCQVT.W. Buffer header requests are enqueued off the MCA Link Table through offset MA.FE. The MCA Link Table has the following format:

Offset		MCA Link Table Definitions
MA.NR	0	Number of I/O Entries.
MA.LR.W	1	Highest link currently in use.
MA.FE	3	First entry (queue of requesting BHs).
MA.LE	17	Highest possible link.

It may be useful to note the various states that the MCA can attain during transmission and reception. The state is stored in the right byte of BH_addr->BQLNK, and in the DCT flag word DCT_addr->DCFLG. The states are the following:

```
STATE0 (0) - State 0, No Timeout
STATE1 (1) - State 1, No Timeout
STATE2 (2) - State 2, No Timeout
STATE3 (3) - State 3, No Timeout, Direct I/O
TIME0  (4) - State 0, Timeout
TIME1  (5) - State 1, Timeout
TIME2  (6) - State 2, Timeout
TIME3  (7) - State 3, Timeout, Direct I/O
```

After the buffer header is enqueued, MCACB returns to its calling routine, UIOENQ. Consequently, UIOENQ returns immediately to its caller, Unit I/O Interface Services (RDB.P, WRB.P, etc.). Since there is no more work to do by the system call interface routine, the call will return to the system call processor. Only the requesting TCB of the caller's process will remain pended.

The C-based algorithm that clearly illustrates the MCACB procedure follows. The only inputs are the unit-specific post-processor address and the unit CCB address.

```
#define STATE3 03                   /* MCA I/O: state 3          */

MCACB (PP_addr, CCB_addr)
{
/*************************************************************
 * This unit's one and only BH was allocated when the unit    *
 * was opened.  Its address was stored in the CCB.            *
 * Save its address locally, and store the CCB addr in the BH.*
 *************************************************************/

   BH_addr = CCB_addr->CBBHR.W;             /* Save BH addr      */
   BH_addr->BQCCB.W = CCB_addr;             /* Save CCB in BH    */
   BH_addr->BQUPD.W = unpend_addr;          /* Save BH pp addr.  */

/*************************************************************
 * Move the unit CCB request parameters into the BH.          *
 *************************************************************/

   BH_addr->BQLNK = CCB_addr->CBDBH;        /* Link number.      */
   BH_addr->BQBLK = CCB_addr->CBDBL;        /* Start block num   */
   BH_addr->BQPRI = CCB_addr->CBPRI;        /* Process PNQF      */
   BH_addr->BQNBK = CCB_addr->CBNBK;        /* Transmission mode */
   BH_addr->BQWDC = - CCB_addr->CBBLN;      /* Word count (neg!) */

   BH_addr->BQMAP.W = CCB_addr->CBPTA.W; /* PTBL address         */
   BH_addr->BQUAD.W = CCB_addr->CBUAD.W; /* User buffer addr     */
   BH_addr->BQRBC.W = 0;                 /* Init run byte cnt    */

/*************************************************************
 * Validate some parameters.                                  *
 * Make sure link number in bounds (0 - 15.).                 *
 * If it is, move it over to the left byte where it belongs.  *
 * If the MCA's last operation was xmit, this must be recv.   *
 * If the MCA's last operation was recv, this must be xmit.   *
 * Also, xmit operation cannot use link 0, else error.        *
 *************************************************************/

   if (BH_addr->BQLNK < 0) || (BH_addr->BQLNK > 15.)
      return (Illegal_link_number)
   else
      BH_addr->BQLNK <<= 8;

   if ( check_bit (BH_addr->BQDCT.W->DCDVC, DCTFLG) )
      {                                     /* Last op receive.  */
      if ( check_bit (CCB_addr, BCBC1) ) /* This op recv too? */
         return (Illegal_I/O_type);         /* Yes, illegal.     */
      }
   else
      {                                     /* Last op transmit. */
      if (!check_bit (CCB_addr, BCBC1) ) /* This op xmit too? */
         return (Illegal_I/O_type);         /* Yes, illegal.     */
      if (BH_addr->BQLNK == 0)              /* No, but is link 0?*/
         return (Illegal_link_number);      /* Yes, gotcha!      */

      if (BH_addr->BQRTY < 0) || (BH_addr->BQRTY > 255.)
         return (Illegal_retry_value);      /* Bad retry count.  */
      }
```

```
/*************************************************************
 * A receive operation was indicated by a set bit in the DCT  *
 * address offset in the buffer header.  Reset it now.        *
 *************************************************************/

   BH_addr->BQDCT.W &= 017777777777;     /* Reset bit 0.      */


/*************************************************************
 * Check transmission mode. (Protocol = 1; Direct I/O = 2)    *
 * If direct I/O, verify that there are no outstanding        *
 * requests.  If there are not, set state 3 in the DCT and in *
 * the status word (right byte of BH_addr->BQLNK).            *
 *************************************************************/

   if (BH_addr->BQNBK > 1)               /* Direct I/O?       */
      {                                  /* Yes.              */
      if (DCT_addr->DCQVT.W->MA.NR != 0) /* Outstanding reqs? */
         return (Protocol_error_251);    /* Yes, illegal.     */

      DCT_addr->DCFLG = STATE3;          /* No, state = 3.    */
      BH_addr->BQLNK |= STATE3;          /* Status = state 3. */
      }

   BH_addr->BQOWC = - BH_addr->BQWDC;    /* Make positive     */
                                         /* word count.       */

/*************************************************************
 * Enqueue the buffer header to the MCA unit.                 *
 * QMCAB does the actual enqueue.  Buffer headers are enqueued*
 * to the MCA link table, offset MA.FE.  The MCA link table   *
 * address is found at DCT_addr->DCQVT (Queue Vector Table).  *
 *************************************************************/

   call MASK (DCT_addr->DCMSK.W);        /* Mask out MCA ints.*/

   call QMCAB (DCT_addr);                /* Enqueue BH to MCA.*/

   call UNMASK;                          /* Allow MCA ints.   */

   return;                               /* Back to caller.   */

} /* end of MCACB */
```

7.9 Enqueuing Unit CCB Requests to Line Printer Units (LPBIO)

The Unit Management service LPBIO extracts this data from the CCB and moves it into the buffer header. LPBIO then readies the buffer header for I/O. LPBIO enqueues the buffer header to the appropriate place.

Since line printers can be hooked up to Unicorn controllers, the buffer header will be enqueued to:

a) the device's DCT request queue, DCT_addr->DCCRQ.W, if a non-Unicorn device, or
b) the device's UDB request queue, UDB_addr->UDCRQ.W, if a Unicorn device.

Like MQCCB, LPBIO invokes the generic Unicorn buffer header enqueue NQUCN, if the LPU is a Unicorn device. Otherwise, LPBIO enqueues the buffer header to the DCT and calls the line printer unit start-up routine to inform the LPU to begin the data channel transfer. The start-up routines for LPB, LPD, and LPE type line printer units are named LPBST, LPDST, and LPEST, respectively. (See Section 7.7 for more on Unicorn units.)

The LPU start-up routines are relatively short and straightforward in comparison to their disk, tape, and MCA equivalents. Therefore, as a matter of both interest and informativeness, the actual assembly language LPBST routine has been duplicated below direct from the LPB device driver. The comments provide insight into the logic.

```
LPBST:  WSSVR   0
        XWLDA   3,DCCRQ.W,2     ; First BH addr on req queue
        IOXCT   DIA             ; Get the status of this unit
        XNLDA   1,DCSDV,2       ; Retrieve it.
        NLDAI   ONRDY,1         ; On-line and ready mask.
        WAND    1,0             ; (Mask)
        SUB#    0,1,SZR         ; Is it both on-line and ready?
          WBR   WTST            ; NO!  Wait for status change.

        NLDAI   BDVFU,1         ; Yes, VFU enable bit (DCT).
        WBTZ    2,1             ; Assume no VFU enable.
        NLDAI   BQEOV,0         ; Get enable VFU bit from BH.
        WSNB    3,0             ; Is the VFU to be loaded?
          WBR   NVFLD           ; No, don't load it.

        WBTO    2,1             ; Yes, set the bit in the DCT.
        WSUB    0,0             ; Send command to LPB to
        IOXCT   DOA             ; load the VFU.
        XNLDA   1,DCSDV,2       ; Status in AC1.

NVLFD:  NLDAI   BDSWT,0         ; Turn off the waiting for status
        WBTZ    2,0             ; bit in the DCT.

        XNLDA   0,BQORC,3       ; Starting byte count.
        NEG     0,0             ; LPB needs it negated.
        IOXCT   DOC             ; Inform LPB.

        XNLDA   1,DCSDV,2       ; Status in AC1.
        WMOV    2,0             ; SWAMP needs DCT addr in AC0
        WMOV    3,2             ; and BH addr in AC2.
        XJSR    SWAMP           ; Set up data channel map.

        WADD    2,2             ; Convert data addr into byte
        WMOV    2,0             ; pointer and move to AC0.
        XWLDA   2,OAC2.W,3      ; Load DCT address,
        XWLDA   3,DCCRQ.W,2     ; and buffer header address.
        XNLDA   1,BQBYT,3       ; Byte flag.
        WADD    1,0             ; Add correction to data addr.
        IOXCT   DOBS            ; Start up the data transfer!
        XNLDA   1,DCSDV,2       ; The device will interrupt
        WRTN                    ; when transfer complete.

; The line printer is not on-line and ready, so we must wait
; for its status to change.

WTST:   NLDAI   BDSWT,0         ; Turn on the waiting for status
        WBTO    2,0             ;   to change bit and wait for
        WRTN                    ;   an interrupt.
```

After the buffer header is enqueued, LPBIO returns to its calling routine, UIOENQ. Consequently, UIOENQ returns immediately to its caller, Unit I/O Interface Services. Since there is no more work to do by the system call interface routine, the call will return to the system call processor (in lieu of calling CWAIT to pend), and only the requesting TCB of the caller's process will remain pended.

The C-based algorithm that clearly illustrates the LPBIO procedure follows. The only inputs are the unit-specific post-processor address and the unit CCB address.

```
LPBIO (PP_addr, CCB_addr)
{
/*************************************************************
 * This unit's one and only BH was allocated when the unit    *
 * was opened.  Its address was stored in the CCB.            *
 * Save its address locally, and store the CCB addr in the BH.*
 *************************************************************/

   BH_addr = CCB_addr->CBBHR.W;           /* Save BH addr      */
   BH_addr->BQCCB.W = CCB_addr;           /* Save CCB in BH    */

/*************************************************************
 * Move the unit CCB request parameters into the BH.          *
 *************************************************************/

   BH_addr->BQMAP.W = CCB_addr->CBPTA.W; /* PTBL address       */
   BH_addr->BQUAD.W = CCB_addr->CBUAD.W; /* User buffer addr   */
   BH_addr->BQQBC = CCB_addr->CBQBC;     /* Req'd byte count   */
   BH_addr->BQPRI = CCB_addr->CBPRI;     /* Process PNQF       */

   BH_addr->BQRBC.W = 0;                 /* Init run byte cnt  */
   BH_addr->BQORC   = 0;                 /* Start byte count   */
   BH_addr->BQBHT   = 0;                 /* Byte flag          */

/*************************************************************
 * Init the BH some more before enqueuing it.                 *
 *************************************************************/

   BH_addr->BQST = QTMOD | QTIOP | QTNAB;    /* BH status     */

   if ( check_bit (CCB_addr, BCBEO) )    /* VFU enable?        */
      set_bit (BH_addr, BQEOV);          /* Yes, set in BH.    */


/*************************************************************
 * Enqueue the buffer header to the LPU device.               *
 * If this unit is a Unicorn device, a special routine must   *
 * handle the buffer header enqueue.  All Unicorn           *
 * device I/O is handled by the same driver (AOS/VS module    *
 * UNICORN.SR).  Refer to Section 7.7 for more details.       *
 * Once the BH is enqueued, we can return to our caller.      *
 * The post-processor will unpend the requestor's TCB.        *
 *************************************************************/
```

```
   DCT_addr = BH_addr->BQDCT.W;          /* Get DCT address.  */

   if ( check_bit (DCT_addr, BDUCN)      /* Unicorn device?   */
      {                                  /* Yes.              */
      call NQUCN (unpend_addr, BH_addr); /* Enqueue BH to UDB.*/
      return;                            /* Nothing else!     */
      }

/*****************************************************************
 * Not a Unicorn device.  Enqueue the BH here.                   *
 * First mask out line printer interrupts, and get DCT lock.     *
 * Enqueue the buffer header to DCT_addr->DCCRQ.W by order of     *
 * the requestor's priority (BH_addr->BQPRI).                    *
 *****************************************************************/

   call MASK (DCT_addr->DCMSK.W);        /* Disallow LPU ints.*/

   BH_addr->BQUPD.W = unpend_addr;       /* PP addr to BH.    */
   set_bit (DCT_addr, BDLOK);            /* Acquire DCT lock. */

   enqueue_by_PNQF (DCT_addr->DCCRQ.W, BH_addr, BQLNK.W);

/*****************************************************************
 * Tell the LPU to start the data transfer.                      *
 * First call UPDT to update BH_addr->BQRBC,BQORC.               *
 * Then, if there is only 1 BH on DCCRQ.W, call the LPU          *
 * start-up routine, which issues commands to the device to      *
 * begin the data channel transfer.  If there was already a      *
 * request on the queue, the LPU is already started.  The        *
 * current request's post-processor will scan the queue and      *
 * force the LPU to run the next request, and so on, until       *
 * the last request is processed.                                *
 *****************************************************************/

   call UPDT (BH_addr);                  /* Update BH fields. */

   if (only_one_BH_on_DCCRQ.W)           /* Start up LPU dev  */
      call DCT_addr->DCSTR.W;            /* not already busy. */

/*****************************************************************
 * The request has been enqueued.  Clean up time.                *
 * Release the DCT lock and allow interrupts on the LPU again.*
 * In congruence with all unit I/O, the requesting process'      *
 * TCB will be unpended by the BH post-processor, so we just     *
 * return.                                                       *
 *****************************************************************/

   clear_bit (DCT_addr, BDLOK);          /* Release DCT lock. */
   call UNMASK;                          /* Allow LPU intrps. */

   return;                               /* Back to caller.   */

} /* end of LPBIO */
```

7.10 Unit I/O Request Post-Processing

After Unit I/O Interface Services enqueues I/O to the unit device, the requesting process' system call completes, but the TCB remains pended. When the data transfer completes, the device generates an interrupt to the host, and the driver services the interrupt. Before the interrupt service routine dismisses the interrupt it calls the (unit I/O) buffer header post-processor.

The main unit post-processor entry points are the following:

PPSUM - MTU/LPU post-processor
PPACM - MCU post-processor

Both of these entry points are defined at the same location; they execute the same code. In reality, these post-processors are "pre post-processors." Since AOS/VS 7.50 will not assume the overhead of unit I/O buffer header post-processing at interrupt level due to time-consuming calls to Memory Management, PPSUM/PPACM must first determine if the JP is running at interrupt level or base level before the "real" post-processing begins. The "real" post-processors do the work of unpending the requesting TCB, unpinning user pages, and moving data into the user's packet. PPSUM/PPACM takes action depending upon the state of the currently executing JP.

If the JP is running at interrupt level, the unit buffer header is enqueued to the global Unit I/O Post-Processing queue, UPPRC.W. Buffer headers are enqueued to the tail of UPPRC.W. (The buffer header is linked through offset BH_addr->BQPPL.W.) Subsequently, the Disk Manager Task is awakened (unpended). The Disk Manager Task, which runs at base level, calls the Unit Management service UIOPP to dequeue all buffer headers from UPPRC.W. For each buffer header on UPPRC.W, UIOPP dispatches to the unique, "real" post-processor routine for the specific unit type. UIOPP dequeues buffer headers from the head of UPPRC.W, which forces a first-in first-out post-processing service.

If the JP is running at base level, PPSUM/PPACM still cannot immediately dispatch to the unit's post-processor. Since unit I/O buffer headers are post-processed on a first-in first-out basis, PPSUM/PPACM must enqueue the current buffer header to the tail of UPPRC.W to preserve the priority ordering. Next, PPSUM/PPACM calls UIOPP to dequeue the buffer headers from UPPRC.W and to invoke their post-processors. (Even if UPPRC.W is empty upon entering PPSUM/PPACM, the buffer header is enqueued anyway, so that UIOPP is always called. This eliminates further checks of UPPRC.W and preserves Unit Management's code modularity.) When UIOPP returns, system-wide unit I/O processing will be complete.

It is important to note that the only type of request that will cause the processor to be at base level in PPSUM/PPACM is a rewind request. The magnetic tape start-up routine issues a rewind request and waits for it to complete. The post-processor is called directly from this routine, which is at base level. The post-processor for all other unit I/O is called from the interrupt service routine.

The following C-based algorithm outlines the unit I/O post-processing:

```
MTAIS ()
{
/*****************************************************
 * The Magnetic Tape Interrupt Service routine       *
 * calls the unit's (initial) post-processor         *
 * routine (for all unit I/O except rewind           *
 * requests).                                        *
 *****************************************************/

                              :
   call *BH_addr->BQUPD.W;
                              :
}


PPSUM ()                      /* MTU/LPU Post-Processor */
PPACM ()                      /* LPU Post-Processor     */
{
/*****************************************************
 * The buffer header is always enqueued to the tail  *
 * of UPPRC.W.  If at base level, UIOPP dequeues     *
 * all BHs and calls their post-processors.  If at   *
 * interrupt level, the Disk Manager Task is         *
 * unpended to call UIOPP from base level.           *
 *****************************************************/

                              :
   enqueue_to_tail (&UPPRC.W, BH_addr, BH_addr->BQPPL.W);

   if (INTLV == 0)            /* At interrupt level?  */
     call UIOPP;              /* No, call pp now.     */
   else                       /* Yes, wake up DMTSK,  */
     call DWAKE;              /* who will call UIOPP. */

   return;
}
```

The "real" unit post-processors are called from UIOPP. These AOS/VS entry points are the following:

MTAPP - MTU post-processor
MCAPP - MCU post-processor
LPBPP - LPU post-processor

The main objective of each of the post-processors is to unpin previously pinned user pages and to unpend the pended user TCB, which is awaiting the I/O completion. User pages are unpinned via the Memory Management service UNPIN. The TCB is readied simply by resetting the TCB pended bit, and the request is officially complete. A sketchy algorithm follows:

```
call MAPUNW (CCB_addr->CBTCB.W);     /* Map the TCB.     */
                :
clear_bit (TCB_addr, ?BTPN);         /* Unpend the TCB. */
                :
call UNMAP (CCB_addr->CBTCB.W);      /* Unmap the TCB.   */
                :
return;                              /* All done!        */
```

8 File Lock Management

8.1 Introduction to File Locking

File Locking was introduced in AOS/VS 7.00. This concept allows cooperating tasks/processes to restrict access to "file elements." In the context of file locking, a "file element" is merely a number that represents a unique lock. There can be a maximum of 2**32 file elements associated with a file. File locking provides for an alternative method of intertask/interprocess communications in relation to gaining access to specific critical regions (e.g., whole files, shared code, database records). The File Lock Management component of the operating system handles all file locking operations.

User processes interface with File Lock Management via two system calls:

?FLOCK - Locks a file element
?FUNLOCK - Unlocks a file element

The corresponding AOS/VS 7.50 File Lock Management services that implement these calls are:

FLOCK.P - Locks a file element
FUNLOCK.P - Unlocks a file element

There are two types of locks: exclusive and shared. Lock requestors have the option of waiting (pending) for a lock or taking the error return from the request if the lock is not available. When an exclusive lock is granted, no other lock requestor can gain access to the element until the lock is released. When a shared lock is granted, all subsequent shared lock requestors are given access to the element until an exclusive lock request is made. In this case, the exclusive lock requestor first pends until the previous shared locks are released, and then gets the exclusive lock. All lock waiters are queued up on a first-in first-out basis.

There is one special type of request, the "whole file" lock request, in which one caller gains exclusive access to all file elements. In order for File Lock Management to grant a whole file lock request, there may be no outstanding file element lock requests and no previous whole file lock requests. No new lock requests are granted until the whole file lock requestor releases the lock.

All file lock requests are associated with a channel. That is, a process must open a file before issuing an ?FLOCK. When the channel is closed, all of the process' file locks that have been granted are released, and all of its pending lock requests are aborted. However, the locks themselves are associated with a file. File lock requests made on different channels, which are open to the same file, compete for the same locks.

File Lock Management is a component of the AOS/VS file system. Nevertheless, let it be clear that files themselves are not locked such that the file is off limits to all other system processes. File locking is only effective when cooperating tasks and/or processes use File Lock Management (?FLOCK/?FUNLOCK) to enforce restrictive access to understood critical regions, and when lock request denials are obeyed.

8.2 File Lock Management Databases

There are four principal Lock Management databases that keep track of all file locking done on a file. All of them are allocated from the File Lock Management database pool, FLOCK.DB. The databases are the following:

1) Lock Management Block (LMB) - one per file
2) Lock Element Block (LEB) - one per element
3) Lock Queue Block (LQB) - one per lock request
4) Whole File Request Block (WFRB) - one per lock request

Lock Management Block (LMB)

Offset		
LMBST	0	LMB lock word.
LMBUC	1	LMB Use Count.
LMBFCB.W	2	FCB address.
LMBFWF.W	4	WFRB queue descriptor (head).
LMBLWF.W	6	WFRB queue descriptor (tail).
LMBHD1.W	10	LEB queue descriptor (head).
LMBTLB.W	12	LEB queue descriptor (tail).
LMBLNG	14	Length of LMB.

When File Lock Management receives the first lock request on a file (first ?FLOCK call), an LMB is allocated. There is one LMB per open file, and its address is stored in the file's FCB at FCB_addr->FBLMB.W. The LMB is deallocated when the last file lock is released (e.g., ?FUNLOCK, implicitly by last close). The LMB use count is incremented upon each lock request (for each LQB or WFRB enqueued), and decremented whenever a lock is released. The LMB database is locked by setting bit BLMBLK (0) in the LMB lock word. It is locked when queue operations are performed. The JP Management service BSLOCK is called to lock the LMB.

Offset		Lock Element Block (LEB)
LEBFLK.W	0	LEB Forward Link.
LEBBLK.W	2	LEB Backward Link.
LEBELN.W	4	Element number.
LEBFQB.W	6	LQB queue descriptor (head).
LEBLQB.W	10	LQB queue descriptor (tail).
LEBSTS.W	12	LEB status word.
LEBUC	13	LEB use count.
LEBLNG	14	Length of LEB.

Associated with every file lock request that attempts to lock a particular element (in contrast to the whole file) is a file element number. File Lock Management allocates an LEB for each element for which there is an outstanding request lock or lock request. Any 32-bit field constitutes a valid element number, which is stored in the LEB. The LEB use count represents the number of requests (LQBs) currently enqueued to the LEB. The only bit pointer in the LEB status word is BLEBLK (offset 0240 from LEB_addr), which represents the LEB lock bit. The LEB is locked when queue operations are being performed. The JP Management service BSLOCK is called to (spin) lock the LEB. Each LEB represents a unique file element number and is enqueued to the LMB.

Offset		Lock Queue Block (LQB)
LQBFLK.W	0	LQB/WFRB Forward Link.
LQBBLK.W	2	LQB/WFRB Forward Link.
LQBWFL.W	4	If WFRB, waiting LEB queue descriptor (head).
LQBWLL.W	6	If WFRB, waiting LEB queue descriptor (tail).
LQBLTP	10	Lock type.
LQBCHN	11	Channel number.
LQBPTB.W	12	PTBL address of locker.
LQBTCB.W	14	TCB address of locker.
LQBTID	16	Unique Task ID of locker (UTID).
LQBLNG	20	Length of LQB.

The database that represents each individual file lock request is the LQB. The LQB is enqueued to the LQB request queue on the LEB of the element it specified. LQBs are enqueued in the order that File Lock Management receives them. Lock requests are granted in this order as well. The requested lock type is stored at LQB_addr->LQBLTP. There are three bit positions defined for this field:

LQTLK (0) - Lock has been granted
LQTSH (14.) - Shared lock
LQTEX (15.) - Exclusive lock

Since only a task that has the file open can issue File Lock Management calls, the channel number is stored in the LQB. The process table address is saved in order to access PTBL_addr->PLCNT, the count of a process' outstanding lock requests (maximum 65,535). The TCB address is needed to access the TCB status word. The Unique Task ID is stored because it is used as an index into PTBL_addr->PTUNL.W. The bit is set in the process table if the process is swapped out and a task is to be unpended upon swap in. This condition can occur if an element becomes free, and the next LQB belongs to a swapped process.

Offset		Process Table Fields involving File Lock Management
		:
PTUNL.W	0146	Bit Mask of tasks to unlock upon process swapin.
PLCNT	0150	Count of outstanding lock requests.
		:

When a whole file lock is requested, an LQB is allocated, but it assumes the name of a Whole File Request Block (WFRB). WFRBs are enqueued to the LMB (not the LEB) because they are associated with the whole file (not an element). In order for a whole file lock request to be granted, there can be no outstanding lock requests (LMB_addr->LMBHD1.W == -1). If there are any outstanding lock requests, the WFRB will be enqueued to LMB_addr->LMBFWF.W, but will pend until LMB_addr->LMBHD1.W

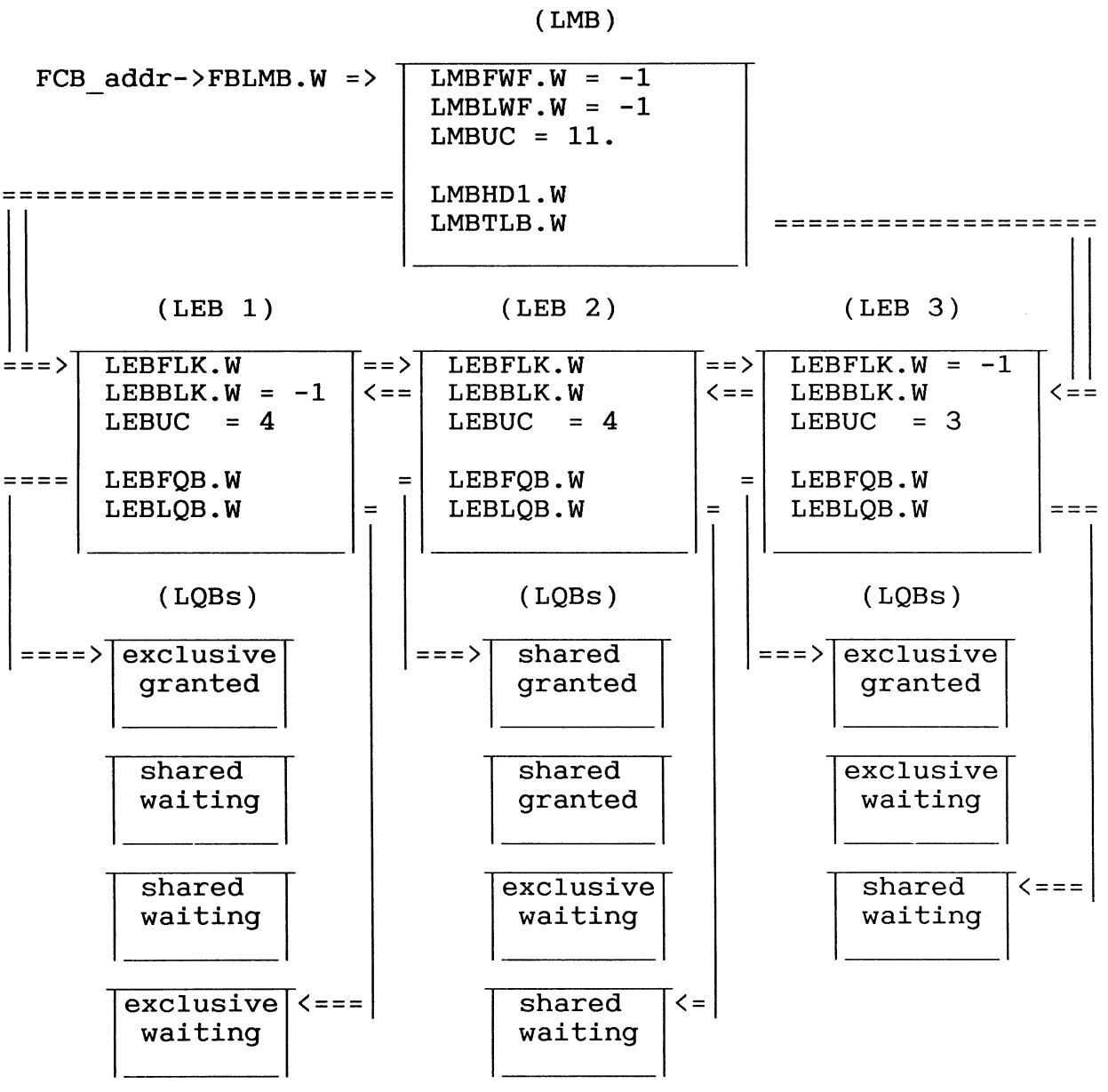
becomes null. File Lock Management uses the same LQB parameters to access the WFRB.

Since all lock requests are serviced in chronological order, all requests that follow a whole file lock request must be enqueued behind the WFRB. Subsequent LEBs are enqueued to the WFRB through the queue descriptor WFRB_addr->LQBWFL.W. When the WFRB releases the whole file lock, the chain of LEBs and LQBs (enqueued to WFRB_addr->LQBWFL.W) is moved to LMB_addr->LMBHD1.W. Those locks are then granted. If another whole file lock request comes through while the WFRB has the lock, but after at least one LEB is enqueued to WFRB_addr->LQBWFL.W, the second WFRB is enqueued to WFRB_addr->LQBFLK.W. When all the LQBs behind the first WFRB acquire and release their file locks, the second WFRB's whole file lock will be granted.

FLOCK.P provides the functionality to allocate and maintain the File Lock Management database structures. FLOCK.P services file lock requests and enqueues the requests to the LMB. FUNLOCK.P provides the functionality to release file locks and to deallocate unused databases.

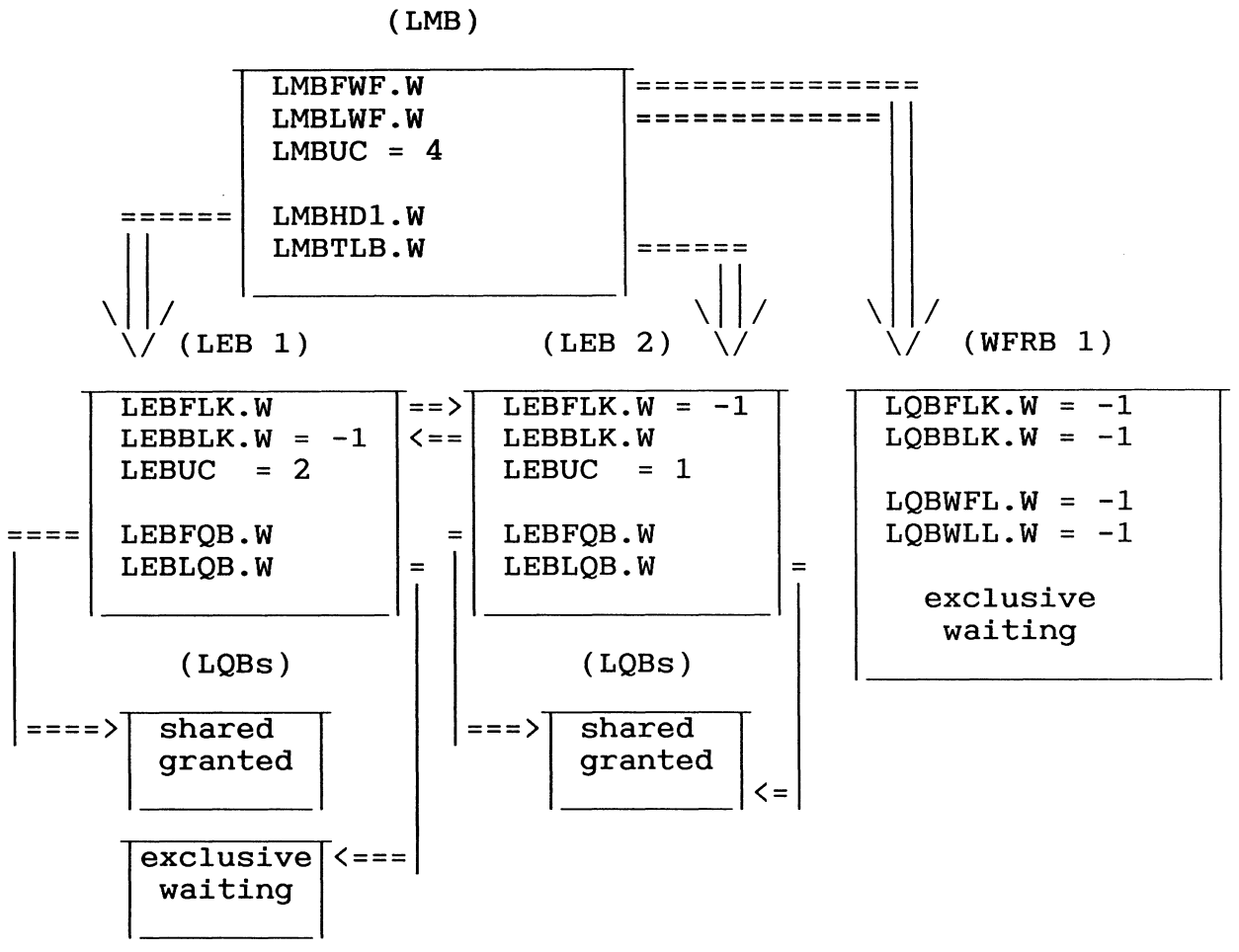
The following illustrations should help to clarify the internal representation of File Lock Management databases which File Lock Management sets up. The diagrams show the links between the LMB, LEBs, LQBs, and WFRBs (of one file) after the prescribed system calls are serviced. The pending mechanism of lock requests is evident as well. Consider the following sequence of events.

1) 4 ?FLOCKs, element 1: exclusive, shared, shared, exclusive
2) 4 ?FLOCKs, element 2: shared, shared, exclusive, shared
3) 3 ?FLOCKs, element 3: exclusive, exclusive, shared



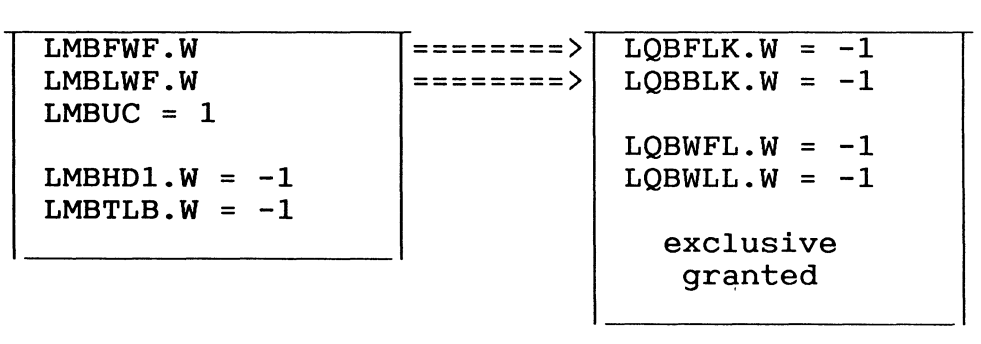
If the first lock request for a given element is exclusive, it will be granted. All subsequent requests to the element, whether exclusive or shared, will pend until the exclusive locker releases the lock. LEB 1 and LEB 3 are representative of this case. If the first lock request for a given element is shared, it and all subsequent shared requests will be granted, until an exclusive request is made. If an exclusive lock request to the element is made, it will pend until all previous shared lock holders release the lock. LEB 2 is representative of this case.

4) ?FLOCK, whole file lock request 1
5) ?FUNLOCKs, element 1: exclusive, shared
6) ?FUNLOCKs, element 2: shared, shared, exclusive
7) ?FUNLOCKs, element 3: exclusive, exclusive, shared

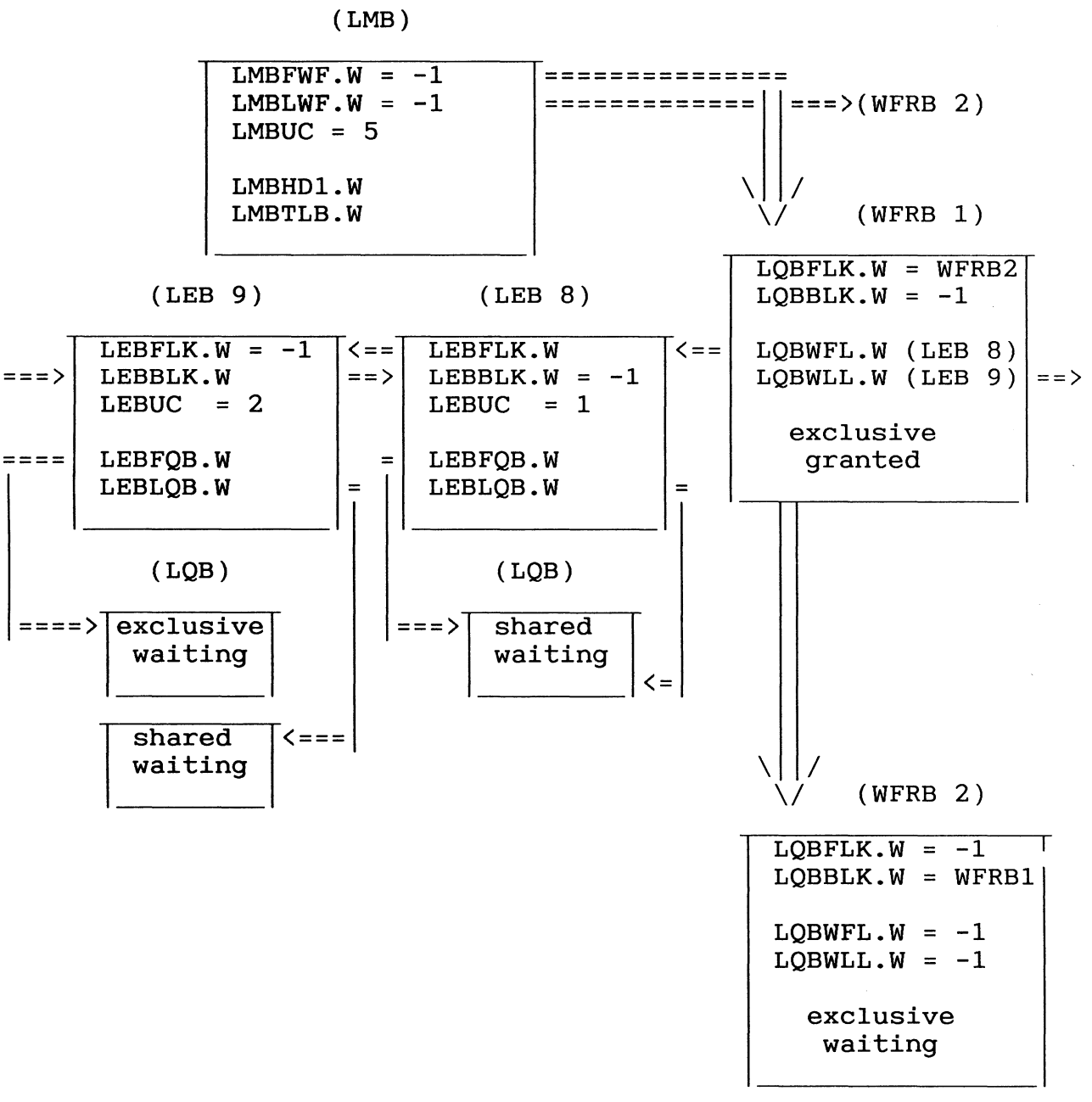


The first two locks on element 1 were released, the first three locks on element 2 were released, and all locks on element 3 were released. Before the whole file lock request can be granted, ALL the locks on LMB_addr->LMBHD1.W must be released. The above diagram shows the databases' current states. If at this point any lock request were made, it would be enqueued off the WFRB. The following diagram shows the databases' states when all file element locks have been released.

8) ?FUNLOCKs, element 1: exclusive, shared
9) ?FUNLOCK, element 2: shared



10) ?FLOCK, element 8: shared
11) ?FLOCKs, element 9: exclusive, shared
12) ?FLOCK, whole file lock request 2



While the whole file lock request is in effect, all subsequent lock requests are enqueued to the WFRB. LEBs are enqueued to WFRB_addr->LQBWFL.W, and WFRBs are linked through WFRB_addr->LQBFLK.W. When WFRB 1 is released in the above diagram, its LEB chain will remain unmodified, but the chain's new queue descriptor will reside in the LMB, namely LMB_addr->LMBHD1.W. Consequently, LMB_addr->LMBLWF.W will point to WFRB 2. Since all locks are serviced in chronological order, all LQB requests must be honored before whole file lock request 2 is granted.