WANG

VS

Data Management System Reference

VS
Data Management System Reference

1st Edition — January 1984

800-1124-01

WANG

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL (617) 459-5000, TWX 710-343-6769, Telex 94-7421

PREFACE

This manual describes the functions of the Wang VS Data Management System (DMS). The Data Management System enables application programmers to create, read, update, and copy data files on a variety of storage media. DMS is system software, supplied to all VS users with each version of the VS operating system, and is used identically on all VS models.

This manual is divided into four parts:

- The first part, Chapters 1 through 3, describes DMS data representation concepts and the structure of DMS records and files. These chapters require no knowledge of a particular programming language.

- The second part, consisting of Chapters 4 through 10, describes how a user program uses DMS to access data in disk files. Chapter 4 provides a general, language-independent overview of data access functions. Chapter 5 provides language-specific overviews of the DMS functions in each of the high-level languages. Chapters 6 through 10 cover these DMS functions in greater detail, using terms and examples from Assembly language.

- The third section of the manual, Chapters 11 through 13, describes the file types and DMS functions for non-disk files. Chapter 11 discusses the use of the workstation screen as a DMS file, Chapter 12 describes DMS processing of files on magnetic tape, and Chapter 13 covers printer files, program files, and word processing files.

- The final section of the manual, Chapters 14 and 15, deals with error processing and special circumstances.

No in-depth knowledge of Assembly language is required to use any part of this manual. However, the latter sections of this manual are directed toward the Assembly language programmer. High-level language programmers should read Chapters 1 through 4 and the section of Chapter 5 describing the DMS features available in their chosen high-level language. From that point, the high-level language user can turn to the specific language reference manual.

Users interested in more in-depth information can refer to the latter chapters of the manual. The high-level language sections of Chapter 5 should provide the necessary background for understanding the details of DMS specified in subsequent chapters.

Chapter 1 provides a general overview of the features available through DMS. Chapters 2 and 3 supply a more detailed view of the structures of records and files. These chapters enable a user to select the most appropriate file and record types for a particular application and to analyze an existing file using the Display utility.

Chapter 4 provides a conceptual introduction to the way that data in files is stored and referenced. The material in these chapters is language-independent.

Chapter 5 provides an overview of DMS functions in all VS-supported languages. Using this chapter, the reader can determine which DMS functions can be performed in a particular high-level language and which must be performed in Assembly language. This chapter describes how to call an Assembly language subroutine to pass DMS parameters from a high-level language. The general syntax for DMS coding in the various high-level languages is included; for additional coding details, the user should consult a specific language reference manual.

Chapters 6 and 7 provide specific information and examples to enable an Assembly language programmer to use file definition parameters and function requests to create and access data files. The scope of these chapters is limited to record access of disk data files; however, many of the features detailed in these chapters are used to access all types of DMS files. Examples are presented in Assembly language.

Chapter 8 explores the Shared File mode. Shared mode is critical when implementing a system in which multiple users must be able to read a file while the file is undergoing modification, or in which two or more users concurrently update a file. This chapter describes DMS file sharing; sharing of DMS/TX files is described in the VS DMS/TX Reference.

Chapter 9 discusses buffering and packing density, which are two ways of improving and maintaining data file performance. These options are especially important when creating large indexed disk files.

Chapter 10 provides Assembly language coding information for the Block Access Method (BAM) and the Physical Access Method (PAM). These access methods are used for processing data by physical units. The Record Access Method (RAM) is described in Chapters 5 through 9.

Chapters 11, 12, and 13 supply the user with the information needed to create and access data files on storage media other than disk. Chapter 11 describes interactive DMS, which uses the workstation screen display as a data file. Chapter 12 deals with access of data files on magnetic tape. Chapter 13 describes specialized files for printers and files that store program information and word processing documents.

Chapter 14 describes DMS error processing and explains how the user can set error addressing options. General classes of error messages are described; for specific error messages, the user should consult Appendix C.

Chapter 15 describes advanced functions, DMS technical features that are only used under certain application-specific conditions. This chapter also describes in detail several advanced programming topics mentioned in other chapters of this manual.

The appendices supply reference material for programming and the analysis of errors. The appendices provide the User File Block (UFB) and Alternate Index Descriptor (AXD1) DSECTs, quick-reference charts to function requests and error codes, the available system-generated GETPARM screens for runtime file assignment and associated Procedure Language statements for assigning values to these screen fields. The program examples within the text demonstrate a particular coding technique, and are not complete programs. Appendix E provides several complete Assembly language program examples.

A detailed index provides assistance in locating material about a specific term or application.

Users of this manual should be familiar with the material contained in the VS Programmers Introduction (800-1101) and the VS Program Development Tools (800-1307) manuals. Familiarity with the COPY, DISPLAY, and SORT utilities is recommended; these are explained in the VS System Utilities Reference (800-1303).

This manual does not describe the DMS file management utilities, which enable users to perform many of the most common DMS operations without writing DMS access programs. These utilities are described in the VS File Management Utilities Reference (800-1308).

DMS data files can be read and updated using the EZQUERY interactive relational query language. This product is described in the VS EZQUERY Concepts and Facilities (800-1337) and the VS EZQUERY Reference (800-1129) manuals.

DMS files can be converted to DMS/TX files by attaching them to a DMS/TX database. Information on DMS/TX file conversion, transaction recovery, and multiple-resource sharing is found in the VS DMS/TX Reference (800-1128).

If additional background or reference material is required, consult the VS Operating System Services manual (800-1107), the VS Principles of Operation (800-1100), or the VS language manual for the appropriate high-level programming language.

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

FIGURES

TABLES

EXAMPLES

EXAMPLES (continued)

EXAMPLES (continued)

CHAPTER 1
DMS INTRODUCTION AND OVERVIEW

1.1 INTRODUCTION

The VS Data Management System (DMS) handles the creation, I/O, and maintenance of all data files on the Wang VS. DMS is used by user programs and system software to access VS data files of all types. You can specify DMS file access on the basis of either physical blocks or logical records.

A working knowledge of the design and functions of DMS allows you to choose the most appropriate file access methods, file storage devices, file and record types, and file I/O functions for an application. You can manipulate data files using the file management utilities (CONTROL, DATENTRY, INQUIRY, REPORT) or the EZQUERY relational query facility without a detailed understanding of the DMS functions supporting these utilities. However, an understanding of DMS allows you to write programs to access data files directly, providing greater control, flexibility, and efficiency in accessing data files. Since all DMS facilities are not available through every VS language, a working knowledge of DMS can help you select the programming language best suited to your data access requirements.

This chapter provides an overview of the major features of DMS, and refers you to chapters that describe each feature in detail. Terms used throughout this manual are introduced in this chapter; they are defined in greater depth in subsequent chapters.

1.2 DATA MANAGEMENT OVERVIEW

Figure 1-1 shows schematically how DMS relates to the other parts of the VS system. This illustration is a logical depiction of how programs access data files; it does not represent the system architecture of any particular VS computer.

A user program issues an instruction to access a file in a programming language. The example shown in Figure 1-1 is a Write instruction, but all file access instructions operate in a similar fashion. The VS treats programs running either interactively or in background, dedicated system tasks, print tasks, and system utilities as equivalent. Each of these supplies file definition parameters and function request modifiers for later use by the DMS.

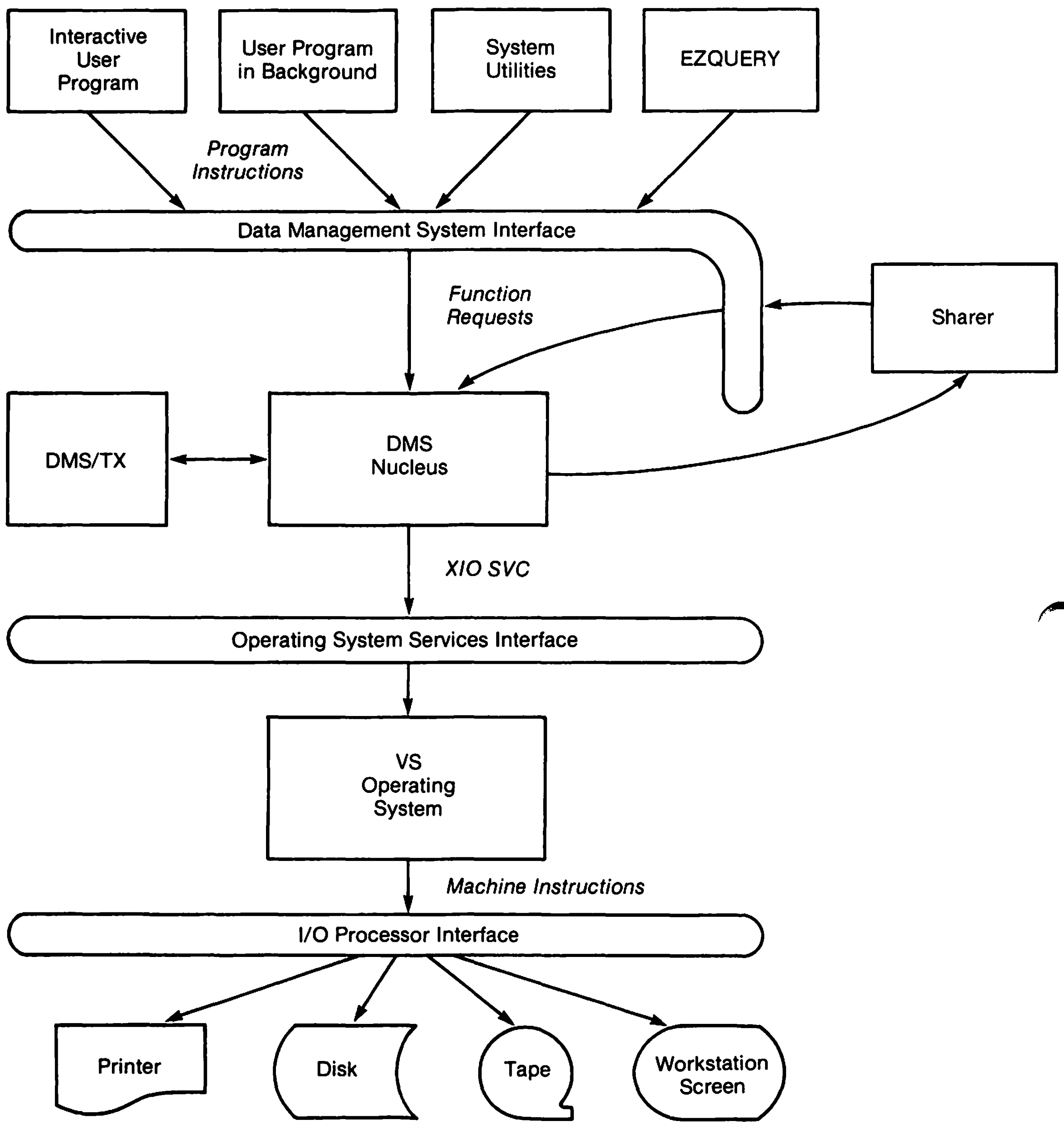


Figure 1-1. DMS Conceptual Overview

DMS Interface

When the program is compiled (or assembled), the system builds file control structures, using either supplied parameter values or system defaults. The file definition parameters that you supply in the program are placed in a control structure called the User File Block (UFB). Other control structures, such as the AXD1, the user record area, and buffer areas are also generated. These control structures act as a DMS interface, translating program I/O instructions into system macroinstructions known as function requests. The DMS interface supplies file definition parameters and function requests to the Data Management System itself. You can interface with this level of functionality directly using Assembly language.

DMS

DMS is system software, included as part of each release of the operating system. It consists of a nucleus of routines that issue supervisor calls (SVCs) to the CPU. The DMS nucleus performs the following seven basic operations:

- The Open routine creates new files and locates existing files. It requires certain file definition parameters to open a file; you supply these to the DMS interface in various ways, including workstation screen interaction or coding them into the user program.
- The Close routine closes a file and releases file resources held by the user. It performs file housekeeping routines and establishes values in certain control blocks for the file.
- The Read function request reads either a logical record or a physical block of data. You can dictate how a read is to be performed by supplying a Read modifier.
- The Write function request writes a record or a physical data block.
- The Rewrite function request modifies a previously written record.
- The Delete function request deletes an existing relative or indexed file record.
- The Start function request performs various pointer positioning functions. You must supply a modifier to all Start function requests to define the specific Start operation.

The Sharer

User programs, EZQUERY, and some utilities can indicate that a file they are accessing should be shared, that the file should remain accessible to other concurrent users. A user program specifies shared processing of a file when it opens the file. If the instruction to open the file indicates that the file is to be shared, DMS issues a message to the Sharer. The Sharer is a dedicated system task that acts an an intermediary for concurrent access to files. When DMS receives a function request for access to a shared file, it issues control instructions to the Sharer. The Sharer coordinating multiple users' access to the file, and issues function requests back to DMS to perform the users' file access requests. Thus, the Sharer is an ordinary task on the system that acts as a proxy, issuing function requests for several user programs.

DMS/TX

Instructions to access a data file are processed through the DMS/TX transaction recovery system if the file requested is attached to a DMS/TX database. DMS/TX maintains before image journals of updated records for the purpose of crash recovery. It also supports multiple resource sharing. User programs access DMS/TX files using standard DMS function request routines. Further information on DMS/TX can be found in the VS DMS/TX Reference.

VS Operating System and Peripherals

To perform function request operations, DMS issues internal supervisor calls (such as the XIO SVC), and reads and writes control block and register values within the VS Operating System. The operating system creates buffers and initiates I/O operations automatically to maximize data file processing efficiency. These internal operations are not described in this manual.

The VS operating system issues I/O requests as machine instructions to peripheral devices (disk and tape drives, workstation screens, and printers) through an IOP or a similar peripherals processor. This peripherals processor controls device specific operations and direct memory access, thus freeing the CPU to perform other tasks.

1.3 RECORD STRUCTURE

DMS supports three record structures: fixed length, variable length, and compressed variable length. All records in a file must be of the same record structure. In most cases, you can select a record structure independent of the file type. With one exception, all file types support all record structures; the relative file type does not support compressed records.

If a file contains fixed length records, all records in the file are of a uniform length. Because the lengths of the file records are the same, the number of records in each block of a consecutive or relative file is the same, with the possible exception of the last block. The location of a particular data record can be calculated from the record length and the record's sequence number (the relative record number).

Variable length records are not of uniform length. Variable length records can range in size from a single byte to an entire block. A maximum record length is specified when the file is created; no record written to the file can exceed this maximum record length.

DMS can compress records in variable length files. The resulting compressed records are always of variable length. DMS performs compression by replacing repetitive characters in a record with a compression code that specifies the repeating character and indicates how many times it is repeated. If you select compression for a file, compression codes are written into all records in the file, whether or not they contain repeating characters. If a character string contains no repeating characters, DMS inserts a code indicating that no compression was possible for that character string. Compression is transparent to the user program. DMS automatically compresses a record when it is written to a file, and automatically decompresses a record when it is read from a file.

For any record structure, the maximum size for a single record is equal to, or slightly less than, one block (2048 bytes). Maximum record sizes vary slightly, depending on file and record type. The longest possible variable length record in a file is 2024 bytes. The longest <u>uncompressed</u> length of a compressed record is also 2024 bytes. Compression saves storage space; it does not allow you to store larger records than would otherwise be allowed in a file.

1.4 <u>FILE STRUCTURE</u>

DMS supports three types of file structures for records: consecutive, relative, and indexed. A consecutive file consists of consecutively written records; the physical sequence of the records in a consecutive file is the same as the chronological order in which those records were written to the file. A relative file consists of consecutive records; a relative file differs from a consecutive file in that records within a relative file can be empty (either never written or deleted). Therefore, you can write records in a relative file either sequentially or by inserting a record in an empty record location ("slot") within the file. The placement of records in an indexed file does not depend on the order in which the records were written to the file, or upon the physical layout of the file. Indexed records are sequenced according to the value of a user-specified key field within each record. DMS automatically maintains pointers to enable the logical sequence of indexed file records to be independent of their physical locations within the file.

Consecutive files are supported on all storage devices. Relative files and indexed files can only be created or accessed on disk storage devices. Relative files are not supported on the VS-50 or VS-80 computers.

Consecutive files can be extended by adding additional records to the end of the file; new records cannot be inserted within a consecutive file. Records cannot be deleted from consecutive files. Records can be read in a consecutive disk file either sequentially or randomly by Relative Record Number. Consecutive files are supported for all types of I/O devices and are used for specialized purposes, such as printer files and system-maintained journals. See Chapter 6 for details on accessing records in consecutive files.

Relative files contain sequential, fixed length record slots. A slot may either contain a record, or it may be empty. You can insert a record within a relative file if there is an empty slot at the desired insertion location. You can also extend a relative file by adding additional records to the end of the file. Records can be deleted from relative files, but the deletion of a record does not reduce the size of the file; it creates an empty slot. You can access records in a relative disk file either sequentially or by relative record number. Relative files are supported on disk only for Operating System Release 6.20 and all subsequent releases. See Chapter 6 for details on accessing records in relative files.

Indexed files contain two distinct types of blocks: blocks containing the actual data records, and index blocks. Index blocks contain entries that are used to locate data records randomly by a user-specified field value. This access field within the data record is known as a key.

All records in an indexed file contain a key field known as the primary key. DMS places data records into indexed file data blocks in ascending order by primary key value. DMS links indexed file data blocks together in ascending order by the primary key values contained within each block, so the logical sequence of records is not dependent upon the physical sequence of data blocks. When you add a record to an indexed file, DMS places the record in a block in primary key sequence and, if necessary, modifies pointer values to sequence the new record in the file according to its primary key value. Records in an indexed file can be updated and deleted. Indexed file records can be read either randomly by key value or sequentially by ascending primary key value.

DMS provides two types of indexed files: primary indexed files, and alternate indexed files. An alternate indexed file is a primary indexed file with additional indices. Unless otherwise specified in this manual, any feature of a primary indexed file is also a feature of an alternate indexed file. Records in both types of files can be read randomly by a unique primary key value. Records in alternate indexed files can also be located by up to 16 additional key fields, called alternate keys. Unlike primary key values, alternate key values do not have to be unique.

Refer to Chapter 3 for the structure of primary and alternate key files. Chapters 5 and 6 describe how the user defines key fields. Chapter 7 describes how a program can be coded for accessing records randomly by key value.

1.5 FILE STORAGE DEVICES

The most powerful and flexible DMS support is available for data files stored on disk. DMS supports sequential file access, random file access by relative record or block number, and random file access by record key values for files stored on disk. DMS provides facilities for multiple users to share data files on disk. Disk is the preferred DMS storage medium and is the device class default. Disks are not, however, the only device class DMS supports. Others include:

WORKSTATIONS	DMS provides interactive access of workstation files. DMS views a workstation screen display as a consecutive data file containing a single 1924-byte record. Special purpose functions are provided for screen definition, and for reading and writing to the workstation screen. See Chapter 11.
MAGNETIC TAPE	DMS supports data file access to data stored on 7- or 9-track magnetic tape reels, or Wang magnetic tape cartridges. Tape is a consecutive storage medium; all tape access is performed sequentially. Tape can be accessed under Record Access Method (RAM), Block Access Method (BAM), or Physical Access Method (PAM) in data transfer units of up to 32K bytes -- far larger than the 2K bytes maximum record and block sizes for disk files. See Chapter 12.
PRINTERS	DMS can write data directly to a printer. Printer files can be output directly to a printer or stored on disk. These consecutive files require special coding for their target device. The special coding requirements for print files, program files, and word processing files are described in Chapter 13.

1.6 ACCESS METHODS

DMS supports three data file access methods: Record Access Method (RAM), Block Access Method (BAM) and Physical Access Method (PAM). RAM is the most commonly used of these, and is the default. RAM accesses data by a file's logical records; that is, records whose length and other characteristics you defined when you created the file. Most of the functions described in this manual are used for accessing logical records within data files using RAM.

BAM and PAM provide faster and more flexible methods of transferring physical units of data. BAM is used to access 2K blocks of data, one block at a time; PAM allows a more flexible access of 2K data blocks, permitting multiple-block transfers, user-designed buffering, and asynchronous processing. Further information on access methods is covered in Chapter 10.

1.7 RUNTIME ASSIGNMENT

To create or access a data file, the user program must first open the file. Files are opened by invoking the Open routine; a file cannot be opened unless you supply certain file definition parameters to the Open routine. How the file definition parameters are coded and what their values should be are described in Chapter 4 and subsequent chapters. This section describes how to supply file definition parameters to the Open routine and what operations the Open routine performs.

Prior to performing an Open operation, you must supply the Open routine with certain items of information necessary to either create a new file or locate an existing file. You can supply these file definition parameters to the Open routine in the following ways:

- Compiled as program statements
- Stored as the user's default parameters
- Provided by PUTPARMs as each file is opened

File definition parameters include the file, library, and volume names, the record and file size, and other parameters. When you create a file, you supply certain file definition parameter values to the compiler (or assembler), which stores these values in an area addressed by the Open routine. If you do not specify a value, some file definition parameters take a system default; others take a default value established for the user running the program. When you access an existing file, DMS retrieves most of the file definition parameter values from the file directory and places them in an area addressed by the Open routine. A few file definition parameters must be specified for both new and existing files. When the information necessary to open a file is not found either in the program, the directory, the system defaults or the user defaults, the Open routine issues a GETPARM.

An Open GETPARM is a request by the system for information; a GETPARM searches for a corresponding PUTPARM. If the appropriate parameter value is found, the GETPARM supplies it to the Open routine. First the GETPARM checks the program's Procedure language instructions. If the appropriate PUTPARM has not been stored as a procedure statement, the GETPARM routine displays a GETPARM screen, requesting the user to supply parameters by typing them at the workstation.

When an Open routine creates a disk file, it uses file definition parameters to calculate the amount of space to be allocated to the new file. This primary allocation is usually composed of a single extent (i.e., a group of physically contiguous blocks of disk space). As file updates add more data to an existing file, the system automatically enlarges the file as needed by adding additional extents to the file.

Under normal conditions, a program terminates processing of a data file by closing the file. When a program issues a command to close a file, DMS releases all system resources associated with that file, thus making them available to other programs. In closing a file, DMS also updates system information about the file. In this way, DMS preserves system integrity and the future accessibility to that file.

1.8 DMS AND DMS/TX

DMS/TX is available to users of Operating System Release 6.10 and subsequent releases. It is an optional feature for DMS indexed disk files that is invoked by the DMS nucleus as part of the Open operation, as shown in Figure 1-1. If you specify DMS/TX support for an indexed file, the system will provide all programs accessing that file with DMS/TX functionality.

DMS/TX allows you to group updates to several records into a transaction. A transaction is either fully applied or not applied at all. If a transaction cannot be fully applied, DMS/TX reverses any updates performed during that transaction. This "rollback recovery" preserves data consistency between files in the event of a system or program crash.

DMS/TX also enables each user to incrementally claim multiple records as needed by the user program, and to hold these records for the duration of a transaction. The multiple resource sharing provided by DMS/TX is superior to that provided by DMS extension rights. While the VS will continue to support DMS extension rights, you are encouraged to code all programs that hold multiple resources in DMS/TX format. You should also convert DMS programs that use extension rights to DMS/TX format at your earliest convenience.

You use the same function requests to access DMS and DMS/TX files. The internal execution of these function requests differs somewhat due to the grouping of file updates into transactions by DMS/TX. For example, a Rewrite function request does not release the rewritten record under DMS/TX. DMS/TX provides additional instructions to define transactions. For efficient processing, the user should include transaction definition instructions in all programs that update DMS/TX files.

Unless otherwise specified, all information in this manual applies to both DMS files and DMS/TX files. For further details on DMS/TX, refer to the VS DMS/TX Reference.

PART I
Data Representation

CHAPTER 2
DATA RECORD STRUCTURE

2.1 INTRODUCTION -- SELECTING A RECORD STRUCTURE

The DMS Record Access Method (RAM) enables you to define logical records within a data file. This chapter describes the available record types, the internal structure of logical records, and their size and placement within physical blocks. The structuring of records into files is described in Chapter 3. Program access to records in disk files is described in Chapters 4 through 7. Workstation, magnetic tape, and printer records are described in Chapters 11, 12, and 13 respectively. Multiple record types are discussed in Chapter 15.

VS DMS supports three types of record structure: fixed length, variable length, and variable length compressed. All fixed length records are allotted the same amount of space in a block. Variable length records can vary in size, based on the record's contents, from one byte up to 2024 bytes. You assign a maximum record length for a variable record file when the file is created. Compressed records are variable length records in which space is conserved by representing repeating characters with a compression code.

For any record structure, the maximum size for a single record is equal to or less than one block (2048 bytes). Maximum record sizes vary slightly, depending on file and record type, as shown below:

Table 2-1. Maximum Record Sizes for File and Record Types

Record Type	File Type	Max. Record Size	Total of All Records
Fixed length	Consecutive	2048 bytes	2048 bytes
Variable length	Consecutive	2024 bytes	2024 bytes
Fixed length	Relative	2040 bytes	2044 bytes
Variable length	Relative	2040 bytes	2040 bytes
Fixed length	Indexed	2040 bytes	2043 bytes
Variable length	Indexed	2024 bytes	2024 bytes

As shown in Table 2-1, the total of the record lengths in a single block can, in some cases, be greater than the size of the largest single record. For example, the largest permitted indexed file record is 2040 bytes. However, if you write multiple indexed records, DMS will block these records to allow the combined length of several fixed length records to be as large as 2043 bytes.

All three record structures are supported in consecutive and indexed files. Relative files support fixed length and variable length records; compressed records are not supported for relative files. Fixed length record structure is the default for all file types. Variable length record structure is most advantageous when a data file is to contain several different record formats with widely differing record sizes. For example, you should establish a variable record structure for a file containing 100-character records and 200-character records. Record compression is recommended when records are expected to contain frequent strings of blanks or other filler characters, or when the nature of the data makes frequent repetitions of a single character common.

You should decide which structure is most appropriate for the length and contents of the records when you first create the file. When you add records to an existing file, DMS stores the added records with the same record structure as the records already in the file. DMS permits records of only one structure in a given data file. You define a file's record structure with file definition parameters, which DMS supplies to the Open operation when DMS initially creates the file. The record type is preserved as a permanent attribute of the file, and DMS automatically formats all records written to the file into the proper record type format. Records that cannot be formatted into the file's record type (e.g., records of the wrong length) are not written to the file. To modify the record structure of an existing file, use the COPY utility to make a copy of the file with the record structure specified by the LENGTH and COMPRESS fields of the utility.

2.2 FIXED LENGTH RECORDS

All fixed length records in a file have the same length, which cannot exceed 2048 characters in a consecutive file, 2040 characters in a relative file or an indexed file. You define the record length when you create the file. This record length is permanently stored in the Volume Table of Contents (VTOC). All subsequent Write operations to the file assume that record length.

DMS enlarges fixed length records in relative files by a two-byte record length indicator. These bytes are located at the beginning of each record; their function is described in Chapter 3.

DMS does not support records spanning a block; if an entire record does not fit into a block, DMS places the record into the next block. DMS places as many complete records as possible in the block, and the remaining disk space in that block is left unused. This results in the same amount of unused space at the end of each block of the file that contains fixed length data records.

As an example, consider a file created by a zookeeper to keep track of the animals in the local Zoological Park. The zookeeper creates one record for each species of animal, giving its name, diet, habitat, etc. Because the same data items must be maintained for every animal in the zoo, the zookeeper stores the data as fixed length records. Each record is 80 bytes in length, and there are 26 records to be placed in the file. The records are stored in 2048-byte (2K) physical blocks. If 2048 is divided by the record length of 80, a quotient of 25 results, with a remainder of 48. Therefore, DMS can store the first 25 records of the file in the first block. When the 26th record is stored, since it cannot fit in the first block (only 48 bytes are left), DMS allocates a second block. The 26th record is stored in the first 80 bytes of the second block. The last 48 bytes of the first block are unusable. The remaining space in the second block can only be used for additional 80-byte records belonging to this file.

File blocks in a consecutive file or a relative file are shown in Figure 2-1. Records in indexed file data blocks are identical, except for the presence of a block length prefix at the beginning of each block and a data link pointer at the end of each block.

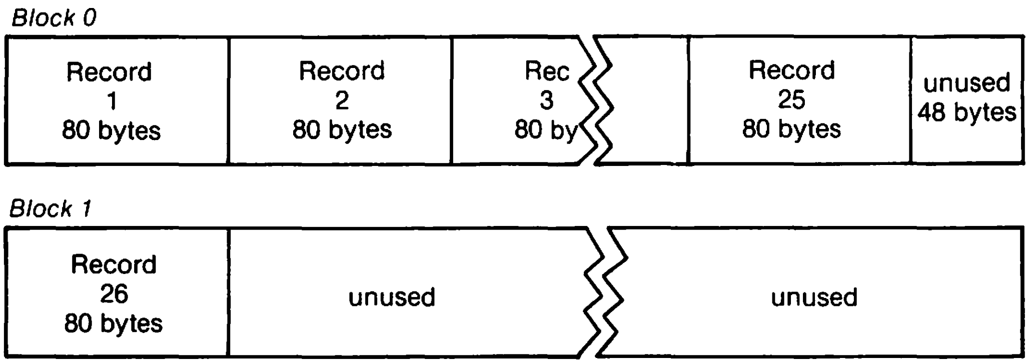


Figure 2-1. Two Data Blocks Containing Fixed Length Records

To minimize unused space in a file, it is important, especially with large records, to establish a record length equal to, or slightly less than a factor of 2048 for consecutive files, 2044 for relative files, or 2043 for indexed files. For example, records of 1024 bytes are stored two per block in a consecutive file with no wasted space. Adding a single byte to the record length, making it 1025 bytes, doubles the number of blocks required for the file. Records of lengths that result in substantial wasted space per block should be restructured or created as variable length compressed records.

NOTE

DMS assigns all alternate indexed records two additional mask bytes per record. This should be considered when you attempt to optimize record lengths. See Chapter 3 for the function of these mask bytes.

The uniform length of fixed length records makes random access of consecutive and relative files efficient. However, it limits the file to a single record length format, and can result in a significant waste of space if the data records contain many repeating characters or trailing blanks. It is difficult to enlarge fixed length records, because to enlarge a single record, you must enlarge every record in the file. In many cases, the user may prefer to use variable length records.

2.3 VARIABLE LENGTH RECORDS

When the difference between a file's shortest and longest record lengths is significant, you should create a file of variable length records. You can conserve considerable storage space in consecutive or indexed files by specifying variable length records. You can specify variable length records for relative files as well, but because these records are written into fixed length slots, no storage space is saved.

Variable length records both save space and simplify data entry. For example, our zookeeper wishes to a maintain a medical history of each animal, in which specific additional fields are added to an animal's record each time the animal is treated by a veterinarian. The zookeeper initially creates the file as a relative or indexed file with variable length records. Making the length of the records variable eliminates the need to add trailing blanks to records to make them all equal in length.

Due to the nature of consecutive files, you cannot rewrite a variable length record with a record longer than the original record. In relative and indexed files, you can rewrite a variable length record with a longer record, if the new record is not longer than the maximum record size you selected when you created the file.

2.3.1 Record Length Indicator

Variable length records can be up to 2024 bytes in length. The user must specify a maximum record length less than or equal to 2024 when creating a file with variable length records. DMS prefixes each record written to a data block with a 2-byte record length indicator (RL), the value of which is equal to the length of the record plus the two bytes of the indicator. For example, an 80-byte record would have a record length indicator with a value of 82. When DMS accesses a record, it first examines the record length indicator then moves the record <u>without</u> the record length indicator to a work area, known as the user record area, provided by the user program.

Relative file records expand the function of this record length indicator by requiring record length indicators for both fixed length and variable length records, and by allowing two indicator values not permitted for other file structures. A record length indicator with a value of 2 indicates the presence of a record with a length of zero. That is, a reserved record slot containing no data. A record length indicator with a value of 0 denotes an empty record slot that is available for use.

2.3.2 Block Length Indicator

Several file and record types begin each block with a 2-byte block length indicator (BL) that indicates the current length in bytes of the contents of the block. Indexed file blocks that contain fixed length records begin with a block length indicator. Consecutive and indexed file blocks that contain variable length records begin with a block length indicator. The value of a block length indicator is the total length of all the records in the block, plus two bytes for the block length indicator itself. For variable length records, the total length of all the records in the block is the sum of all the record length indicators. As in the case of fixed length records, unused space may exist at the end of a block of variable length records. If the next record cannot fit into the remaining space in the block, that space is left unused and another data block is allocated.

Relative file blocks do not contain a block length indicator. The data block format for variable length records in consecutive files and indexed files is illustrated in Figure 2-2. This figure does not show the Data Link Pointer (DLP) field found at the end of indexed data blocks, and the 2-byte mask field appended to all alternate indexed records. See Chapter 3 for an explanation of these fields.

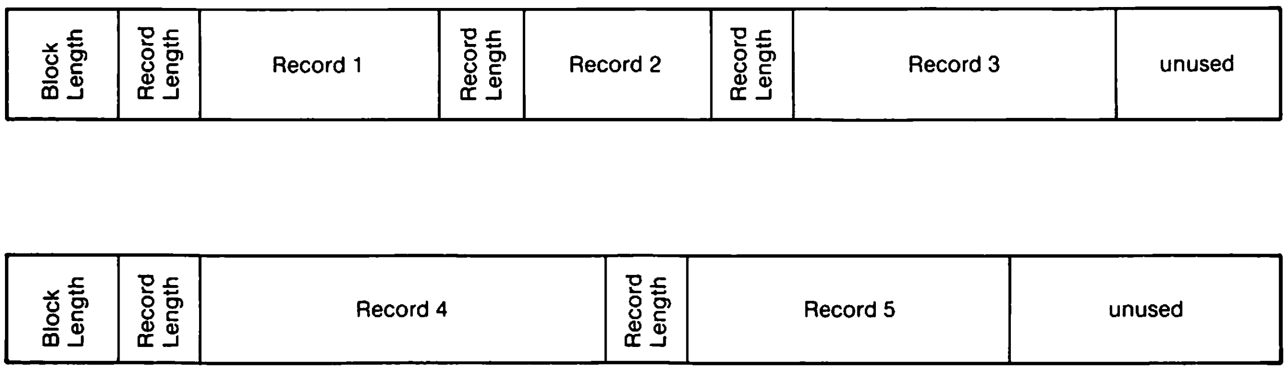


Figure 2-2. Two Data Blocks Containing Variable Length Records

2.3.3 Processing Variable Length Record Files

Sequential processing of variable length records is not as rapid as the processing of fixed length records because DMS must locate and read each record length indicator. Because all relative records contain a record length indicator, there should be no difference in relative file performance in processing records with these two structures.

You can update variable length records in consecutive files by rewriting an existing record if the original record was not compressed and the new record is the same length as the original record. These restrictions do not apply to relative files or indexed files.

You can expand variable length records in indexed files after creating them by adding new fields or enlarging existing ones. If future file expansion is likely for an indexed file, you should initially block the file to allow for expansion, by using a data packing density of less than 100%. For example, if the packing density is set at 50%, DMS uses only half of each block for the initial writing of records; it retains the other half of the block to provide room for expansion of the block's records or for additional records. Packing density is described in greater detail in Chapter 9.

2.3.4 Longest Anticipated Record

Variable length record processing requires specifying two record lengths: the longest anticipated record length for all file records, and the actual length for each record in the file. The longest anticipated record length serves as a maximum length for records written to the file. The user program specifies the longest anticipated record length as the record length parameter when the file is created. DMS saves this parameter value as a permanent attribute of the file.

The program must also indicate the length of each individual record. Before a program that creates variable record writes each record, it must place the length of that record in the record length parameter field. DMS then compares the longest anticipated record length with the length of the current record. Records longer than the longest anticipated record length are not written to the file.

When a record exceeding the longest anticipated record length is input, DMS returns a file status code indicating this condition (File Status '97' or '84'), and checks for the presence of an error routine in the user program. This user-supplied error routine specifies what happens to the input record and the data file. If the program does not provide an error routine, a fatal error cancels the program and closes all open files. Error routines are described in greater detail in Chapter 14.

2.3.5 Primary Disk Space Allocation for Variable Length Record Files

DMS assigns disk space based on the number of records to be written, and the record length specified when the file is created. This disk space assignment is known as the primary allocation. To assign disk space for fixed length records, or for variable records in relative files (which are stored in fixed length record slots) DMS performs a simple calculation based on the record length and the number of records.

However, because DMS cannot know the actual record lengths of variable records in consecutive or indexed files until the records are written to the file, it computes the length of the primary allocation based on the longest anticipated record length. If the difference between the longest record and the average record is significant, DMS may allocate more disk space than is needed for the file. This extra space can be released when the file is created by specifying RELEASE=YES on the GETPARM screen used to define that file (see Chapter 6). If enlargement of the file is anticipated, it may be desirable to retain these extra blocks, rather than releasing them.

2.4 COMPRESSED RECORDS

You have the option of selecting record compression when creating a consecutive or indexed file containing variable length records. Records in relative files cannot be compressed. Selecting record compression causes DMS to compress the contents of all records placed in the data file by representing strings of repeating characters with a compression code. This option can result in greatly reduced file storage requirements; the actual space saved is dependent on the contents of the records. If the variable length records contain numerous repeating blanks or characters, you should select compression.

Compression can speed data transfer. A compressed file can often contain more records per block than an uncompressed file containing the same data. Because fewer blocks means that fewer I/O operations are required to read a file, compression can significantly speed throughput when records are processed consecutively. Transmission time for VS to VS telecommunications can in many cases be significantly reduced by the use of compressed records. Telecommunication emulation (2780, 3780, TTY) is not affected by file compression because files are uncompressed prior to emulation conversion.

NOTE

Compression does not invariably result in shorter records. If a file's records contain few repeating characters, compressed records can actually be longer than the corresponding fixed length records. Use the COPY utility to create a compressed version of a file for comparison.

Generally, the records most amenable to compression contain fields with many blanks or zeros. A workstation screen image, for example, is usually a 1924-byte fixed length record, and most of those bytes are blanks. In order to facilitate displaying of workstation screens, DMS does not compress screen images. However, a user program that copies screen images to a disk or tape file would function more efficiently as a compressed file than as a file with 1924-byte fixed length records. When DMS stores print records on disk, they are always compressed.

2.4.1 Compressed File Processing

You specify compression as a file definition parameter when you initially create the file. Any file you designate as compressed must also be specified as containing variable length records. All records in a compressed data file undergo compression processing, even if no repeating characters appear in the record.

DMS performs compression when data is written into a data file buffer block, and expansion as a record is read from a buffer block to the record area defined by the program. The actual compression processing is performed by the COMP and XPAND machine instructions. (Refer to the VS Principles of Operation, Chapter 7, for further details.)

The largest uncompressed size of a record to be compressed is 2024 bytes. DMS imposes this limit on the size of the uncompressed record to avoid overflowing a file block. In a worst case situation, compression of a 2024-byte record could result in the actual enlargement of the record to greater than 2024 bytes, due to the inclusion of compression codes. Although compression generally reduces the space requirements of a record by 25% to 50%, this reduction cannot be assumed for every record in a data file. Compression cannot be used to create records larger than the size of a block. Compression generally results in more records stored in each block than in the corresponding uncompressed file.

2.4.2 Block Structure of Compressed Records

Compressed records have the same block format as variable length records. The record length prefix to each record contains the compressed length of the record; that is, the actual space it occupies in the block. DMS determines the uncompressed length of a record as it performs a Read operation on the record. DMS places the uncompressed length of the current record in the file's RECSIZE parameter field.

DMS only compresses data; the Record Length (RL) and Block Length (BL) indicator bytes are never compressed; nor are alternate index masks, data link pointers, and index blocks. Primary and alternate index keys are compressed in data records but not in index blocks.

2.4.3 The Compression Code

Compression causes any string of 3 to 128 repeating characters to be stored as a compression code. DMS automatically inserts these compression codes into the contents of the data record, replacing repeating characters. DMS performs this operation on data records individually, before the data records are written to a data file block. As a consequence, record length and block length indicators are never compressed; the first character of a compressed record after the record length indicator is the first compression code byte.

The compression code consists of a compression byte followed by a character byte. The compression byte specifies whether or not, and for how many bytes, compression is to occur. The high order bit of the compression byte contains either a Binary 1 for compression or a Binary 0 for no compression. This bit effectively turns compression on or off for the number of bytes (up to 128) specified by the other seven bits of the compression byte. In hexadecimal, a value of 80 or more indicates compression.

The seven low order bits of the compression byte contain the number of character repetitions. This number is equal to one less than the number of instances of the repeating character in the uncompressed string. Seven bits can contain an integer up to 127; therefore, a compression byte can indicate the compression (or non-compression) of up to 128 characters. Any string longer than 128 characters requires additional compression codes.

The character byte appears immediately after the compression byte. The character byte value is the repeating character. This byte can take any value, including a blank or an undisplayable character. Compression is not limited to the ASCII character set; any repeating byte value can be compressed.

The character byte is not included in the repetition count of the compression byte. For example, a string of six "R"s is represented as a compression byte with a repetition count of five followed by the repeated character byte. This is represented in hexadecimal as follows:

```
        hex '52 52 52 52 52 52'   compresses to   hex '85 52'
```

2.4.4 Character Strings not Requiring Compression

All character strings in a compressed record are delineated by compression codes. When a character string in a compressed record does not contain three consecutive repeating characters no compression is performed on the string. However, the string must still have a compression byte indicating the beginning and length of the non-compressed string. The first bit of the compression byte is set to zero, indicating no compression. The seven compression count bits indicate the number of characters (minus 1) that do not require compression. The uncompressed character string follows the compression byte.

```
  WXYZ   =   hex '57 58 59 5A'   compresses to   hex '03 57 58 59 5A'
```

If the system encounters a string of more than 128 non-compressible characters, it must insert a second compression code after 128 characters indicating further non-compression. The number of repetitions specified by the compression count is one less than the number of uncompressed characters in the string. If an entire record contains no compressible character strings, it must still contain a compression code every 128 characters. Thus, a compressed record can be longer than the same record in fixed length format.

2.4.5 Locating and Interpreting Compression Codes

If it is necessary to inspect a block of compressed data, compression codes within a record can be located by sequentially interpreting each compression code from the start of the record, and counting the number of characters represented in the compression count. The location of compression codes should be calculated; a compression code cannot reliably be located by its value. As an example of reading compression codes, a record containing the character string "GRRRRRREAT" is represented in compressed hexadecimal format as shown in Figure 2-3.

0 0	**4 7**	**8 5**	**5 2**	**0 2**	**4 5**	**4 1**	**5 4**
no compression	"G"	compress	"R"	no compression	"E"	"A"	"T"

Figure 2-3. Hexadecimal Representation of a Compressed Character String

The compressed string shown in Figure 2-3 can be interpreted as follows:

Byte Value	Explanation
0 0	The 0 in this character's high-order bit denotes that the string that follows is uncompressed. The zeros in the remaining seven bits denote the length of the uncompressed string as 1 character (the repetition count of 0, plus 1). Therefore, the next compression code should be the third character of the string.
4 7	The 1-character uncompressed string has a value of "G".
8 5	This byte indicates compression, because the high-order half byte has a value of eight or greater (the high-order bit is set). The seven low-order bits of the byte give the repetition count: 000 0101, or 5 in decimal. Therefore, the character in the following byte is repeated 5 times.
5 2	In uncompressed format this character ("R") would appear six times -- an initial appearance and five repetitions as specified in the preceding byte.
0 2	This byte indicates non-compression, since the high-order bit is a zero. The seven low-order bits indicate one less than the total non-compressed characters that follow -- in this case a value of 2, indicating that three uncompressed characters follow.
4 5	This byte contains the character "E".
4 1	This byte contains the character "A".
5 4	This byte contains the character "T".

2.4.6 Compression Recommendations

The compression option is clearly useful for files in which there are many repeating data characters. However, in files in which there are no or very few repeating characters, compression may not significantly reduce storage requirements, due to the presence of the compression bytes. In fact, the compression option may cause a file to be larger than it would be without record compression, because in a compressed record each uncompressible string adds a byte to the length of the record. It is therefore a good idea to compare the compressed and uncompressed lengths of sample file data before selecting compression. This can be done using the COPY utility.

CHAPTER 3
FILE STRUCTURE

3.1 OVERVIEW

DMS stores records on disk in data files with consecutive, relative, or indexed file structure. Indexed data files are further divided into two types: primary indexed files and alternate indexed files. You can store records of any structure in any type of file, with the exception that compressed records cannot be stored in a relative file. You should select the file structure best suited for your particular application. You can choose any of the following DMS file structures:

Consecutive	Allows you to access records sequentially, and read fixed length records on disk directly by record sequence number. Records can only be added at the end of the file, and cannot be deleted. This structure is appropriate for most data entry and batch update applications, and is the only file organization that is supported for all device types.
Relative	Allows you to access records either sequentially or directly by record sequence number. You can add or delete records within a relative file. However, you must preallocate space for adding records; deleting records does not reduce the size of the file. You should choose a relative file structure if speed of access and the ability to modify and delete existing records is a major consideration. Relative files are only supported on disk. Relative files are not supported on the VS-50 or VS-80 computers.
Primary Indexed	Allows you to access records through a key field that contains unique data values. This structure supports sequential record retrieval, and rapid non-sequential retrieval of single records from disk files by key value. You can add, update, or delete records by specifying the primary key value of the desired record.

Alternate Indexed	Offers all of the features of primary indexed files, as well as allowing non-sequential record access by up to 16 alternate key fields. Thus, you can establish several fields within each record for record retrieval by the value of the data in those fields. Unlike the primary key field, these alternate key fields can contain duplicate data values. Alternate indexed files are well suited to interactive data retrieval from disk data files.

This chapter describes consecutive, relative, indexed, and alternate indexed file structures in detail. Using this chapter and the DISPLAY utility, you should be able to identify the file structure of a data file, and locate the different types of blocks, pointers, length indicators and keys in a data file. This chapter does not describe how to create and access a data file of a particular structure. Material on RAM disk file creation and access is found in Chapters 4 through 7. Magnetic tape file access is described in Chapter 12.

3.2 CONSECUTIVE FILES

Consecutive disk files consist of sequential data records; that is, records stored in the order in which they are created. DMS supports both sequential and random access (also known as direct access) for sequential record retrieval. When reading sequentially, a program reads records in the file in the order written. Random access allows the program to read (using a read relative statement) a particular record in a consecutive file by specifying that record's sequence number (record sequence numbers start with 1).

Consecutive file structure is the default when creating a file. It is the only file type that can be accessed from magnetic tape. Log files, workstation screen files, and printer files use the consecutive file structure. DMS supports sequential access for all device types; DMS only supports random access for consecutive files on disk.

Updating records in a consecutive file is limited in several respects. You can modify fixed length records in consecutive disk files by locating and overwriting individual records within the file. Modification of variable length records on disk is supported if the record length is not changed by the modification; modification of compressed records is not supported. Record modification is not supported for magnetic tape files. Records cannot be deleted from consecutive files.

You can process consecutive files using all three file access methods: RAM, BAM, and PAM (see Chapter 10). Each of these access methods supports several processing modes and function requests (see Chapters 6 and 7). These features make consecutive files particularly well suited to such applications as transaction recording, in which records are stored sequentially in the transaction file and then sorted into the master file.

3.2.1 Space Allocation for Consecutive Disk Files

When you create a consecutive file on disk, DMS uses the record length and the estimated number of records to calculate the file's initial space requirement. This space requirement is called the primary allocation. The primary allocation can be as large as three extents. An extent is a group of physically contiguous 2K-byte blocks on disk. If you extend a file so that it exceeds the primary allocation, DMS automatically assigns additional extents as needed up to a maximum of thirteen. Each additional extent is approximately half the size of the primary allocation. Because the number of extents in the primary allocation depends on the sizes of available extents on a disk, the same consecutive file may require a different number of extents when copied onto another disk pack. If DMS has allocated thirteen extents to a data file, and further file space is required, DMS terminates the operation requesting additional space with a file status code 34 (boundary violation condition).

You can extend a consecutive file by opening the file in Extend mode. Extension can enlarge a consecutive file up to a maximum of thirteen extents. This is approximately six times the length of the original file, if the primary allocation was a single extent. (See Chapter 6 for an explanation of Extend mode.)

3.2.2 Buffering for Consecutive Files

DMS writes records into consecutive files by moving the records into a main memory buffer, then copying the buffer block to the file. DMS moves each record individually from the user record area to the user buffer. When a block of records has accumulated in the buffer, DMS copies the buffer block to the file as a single unit.

You can set the size of this buffer to any multiple of 2K bytes up to 18K bytes. Because using a larger buffer block reduces the number of physical I/O (XIO) operations, the use of a large buffer can speed record retrieval for consecutive files. Refer to Chapter 9, Efficiency Considerations.

3.3 RELATIVE FILES

A relative file is in many respects similar to a consecutive file. A relative file consists of sequential records for which DMS supports both sequential and direct access. When reading sequentially, a program reads the data records in the file in the order of their physical sequence in the file. Direct access allows the program to access a particular record in a relative file by specifying that record's sequence number (record sequence numbers start with 1).

The principal difference between consecutive files and relative files is that in a relative file the space for a record can be created without actually placing any data in that record space. You can, subsequently, use these empty record slots as places to insert records within a relative file.

Because access of records in a relative file is direct, any record in the file can be accessed in a single I/O operation. This makes relative files an extremely efficient file structure for random retrieval of data records. You can process relative files using all three file access methods: RAM, BAM, and PAM (see Chapter 10).

All record slots in a relative file are of fixed length. The length of the data within the record slot can be variable, from zero bytes up to the maximum record size. The first two bytes of every record slot are a system-generated record length field which DMS uses to determine if a slot is empty or if it contains a record. Figure 3-1 shows a relative file containing variable length records.

00	**28** ABCDEFGH IJKLMNOPQR STUVWXYZ	**02**	**00**	**05** ABC

Figure 3-1. Relative File Structure

In the relative file shown in Figure 3-1, relative records number 1 and 4 are empty record slots (record length = 00). Relative record numbers 2 and 5 contain variable length records. The record length indicates the length of the data plus the two-byte record length indicator. Relative record number 3 is a zero-length record. The record length indicator shows that a record is present, but that record consists of only the two-byte record length indicator.

When reading a relative file sequentially, DMS skips over empty record slots, and only reads actual data records. When accessing a relative file directly, you can read, rewrite, and delete records in record slots that already contain data. You can write a record to a relative file by placing it in an empty record slot, or write the record to the end of the file.

If the relative file contains variable length records, you can modify a record by locating and overwriting the record with a record of equal, greater, or lesser length, as long as the record length is not greater than the maximum record length for the file. Compressed records are not supported. Records can be deleted from relative files; deleting a record leaves an empty record slot available for the addition of a new record.

3.4 INDEXED FILES

Indexed disk files are of particular importance to the programmer, because records in an indexed file can be accessed either sequentially or randomly by a key field. These methods provide you with the flexibility to tailor record retrieval to the application. DMS performs random access to an indexed file by using a key field within each record; it locates a particular key value by means of a tree of index blocks. The use of a keyed structure enables you to rapidly retrieve individual data records without knowing their physical locations within the file.

You can update indexed files on disk by inserting new records between existing records without regard for the physical layout of the file. In addition, you can delete records from indexed files. These features make indexed file structure especially appropriate for frequently updated disk data files.

Indexed files require greater disk space than consecutive files, and offer no advantage in speed of access when read sequentially. Therefore, you should establish files as consecutive or relative, unless there is a specific reason for making them indexed.

3.4.1 Data Block Structure for Indexed Files

There are two types of indexed files, primary indexed and alternate indexed. Alternate indexed files have all the structural features of primary indexed files, along with some additional features. This section describes the structure of a primary indexed file, illustrating those features common to all indexed files.

Indexed disk files consist of two types of 2K blocks: data blocks and index blocks. The majority of the blocks in an indexed file are data blocks that contain the actual data records. DMS maintains a pointer to the first data block of an indexed file. The first data block in a primary indexed file is usually, although not always, Relative Block zero.

One or more index blocks are found in every indexed file. DMS initially places these index blocks in the center of the first extent. DMS uses these blocks for keyed access to records in data blocks. Index blocks are only used for random access of individual records; they are not used for sequential file access.

NOTE

If a file has been formatted for DMS/TX processing, the first two blocks of the file are File Recovery Blocks. File Recovery Blocks are user-transparent, and are ignored by user programs and utilities such as DISPLAY. These blocks are described in the VS DMS/TX Reference.

All blocks in indexed files begin with a 2-byte block length indicator. DMS creates this block length indicator for fixed length as well as variable length record data blocks, and for all index blocks. The value of this indicator contains the number of bytes of information within the block, including the 2 bytes occupied by the block length indicator.

The last three bytes of every block (bytes 2046-2048, counting from 1) in an indexed file contain the relative block number of the next logically consecutive block. A logically consecutive data block contains the next record in ascending primary key sequence. For data blocks, this relative block number field is called the Data Link Pointer (DLP). For index blocks, the logically sequential block is the next index block in ascending primary key sequence that is on the same level of the tree structure. The index block pointer at the end of each index block is called the Key Link Pointer (KLP). The final block in each sequence contains high values (hex FFFFFF) in its last three bytes. An indexed file tree structure, showing link pointer values is shown in Figure 3-2. Index block structure and function are described in greater detail in later sections of this chapter.

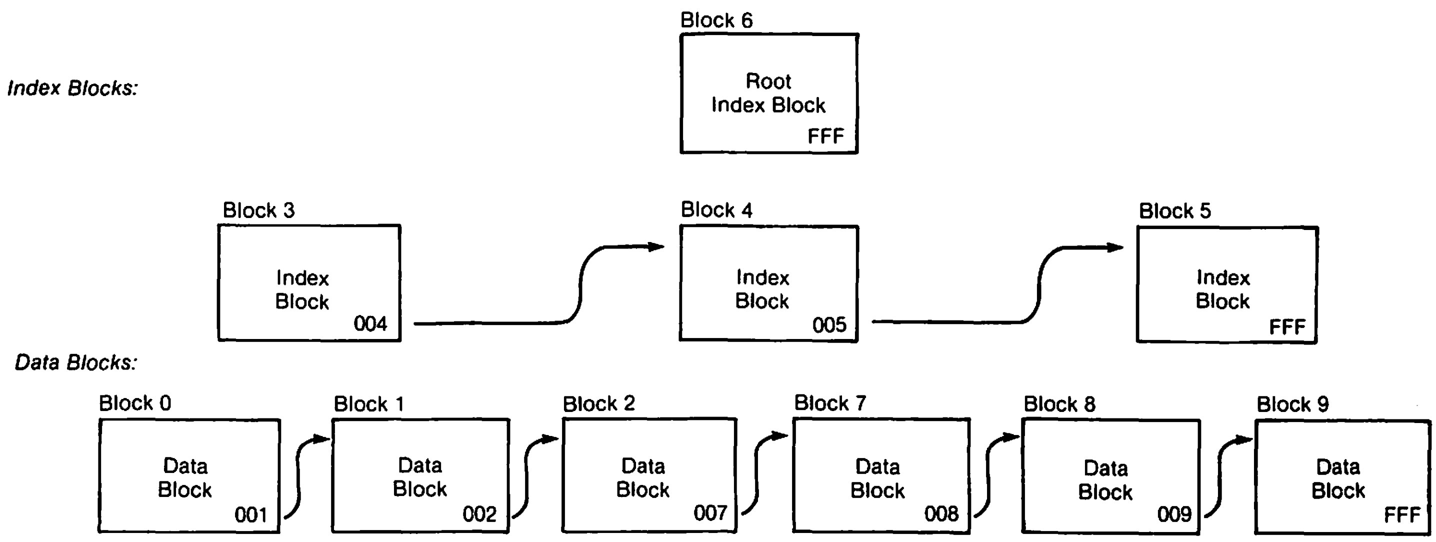


Figure 3-2. Indexed File Showing DLP and KLP Values

3.4.2 <u>Sequential Access of Indexed Files</u>

You can access indexed files sequentially by opening the file and issuing read statements. The address of the first data block is located in the FDR1 record of the disk's Volume Table Of Contents (VTOC), and all subsequent data blocks are linked together by three-byte data link pointers (DLP) at the end of each data block, as shown in Figure 3-2.

You can initiate sequential access at any record within the file by specifying a key value with which to begin sequential access. DMS locates the first record by key value; subsequent Read operations locate records sequentially from that point using the data link pointers. For more information on the Read function request consult Chapters 5 and 7.

DMS carries out sequential access of an indexed file by using the data link pointer at the end of each data block to locate the next logically sequential data block. It does not necessarily store logically sequential data blocks as physically sequential blocks, because index blocks are also present in the file, and because updates to the file can move logically consecutive records into blocks far removed from one another (see Section 3.4.7, Block Splitting). For these reasons, DMS uses pointers to chain data blocks together. A 3-byte data link pointer in bytes 2046 to 2048 (counting from one) of each block contains the relative block number of the next data block.

The following example illustrates the use of an indexed file. A zookeeper wishing to maintain rapidly accessible records for each type of animal might store information in an indexed file that uses animal names as a key field. Because DMS stores records in ascending order by primary key, the animal records are stored in alphabetical order by name. Typical data blocks would appear as shown in Figure 3-3.

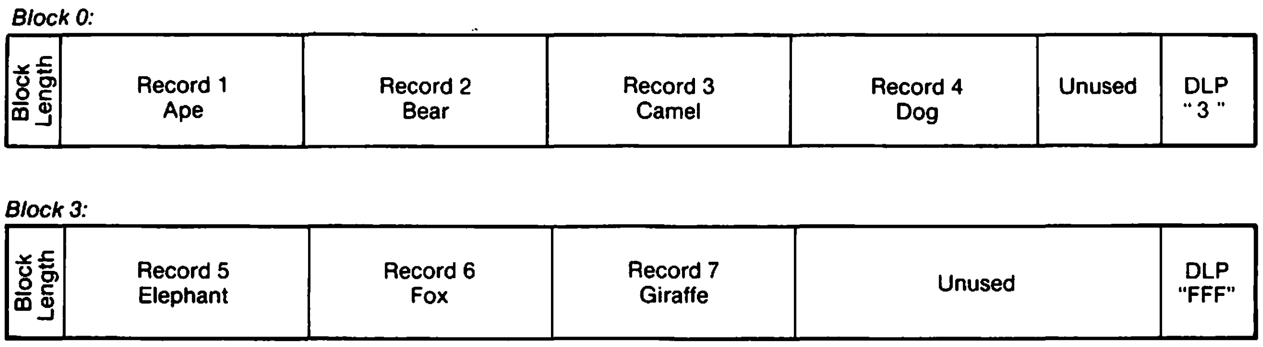


Figure 3-3. Indexed File Data Blocks

The Data Link Pointer (DLP) allows the sequential seeking and reading of records from block to block, even (as in the case above) when the data blocks themselves are not consecutive. The final data block contains high values (represented in hexadecimal as FFFFFF) in the data link pointer, to indicate the end of the file.

Due to the space requirements of the data link pointer and block length indicator, the maximum record size for fixed length records in an indexed file is 2040 bytes.

3.4.3 Primary Key

Every indexed file data record contains a primary key. This key is a fixed length field containing a unique value used for storing and retrieving records. DMS stores records in ascending primary key sequence, using the ASCII collating sequence. A primary key can be up to 255 bytes long in any combination of character types. The following are some guidelines for selecting a primary key field:

1. All primary key values must be unique. You should select a key field value that will always be unique (such as social security numbers), rather than one that is only unique for all current data (such as first and last names).

2. You cannot change the primary key value after assigning it, except by deleting the record and creating a new record. Therefore, you should select a key field value that is invariable as well as unique. For example, a person's telephone extension would not make a good primary key field.

3. You cannot enlarge the primary key, except by copying the entire file. Therefore, you should avoid a primary key that can be exhausted (such as a three-digit employee number).

4. The primary key should be universal. All possible additions to the file should have a value for the primary key. For example, a company with offices only in the United States can assume that all of its employees have Social Security numbers. However, if the company expands internationally, it can no longer make this assumption. Therefore, Social Security number would not be a good primary key for the employee files of an international organization.

5. Records are sequenced in the file by primary key. Therefore, you can use the first character(s) of a primary key to group records together that are often accessed as a group. You can use this technique to maximize the use of sequential access, to improve buffer efficiency in random access, and to allow you to hold related records by a generic key.

6. Primary key access is more efficient than alternate key access. Therefore, you should establish the unique identifier most commonly used for record retrieval as the primary key.

7. You should make the primary key as short as possible to conserve space and improve performance.

A key field can include more than one contiguous data record field. For example, if our zookeeper wanted to maintain separate records on males and females, the primary key might include the animal name field and the sex field of each record, as shown in Figure 3-4.

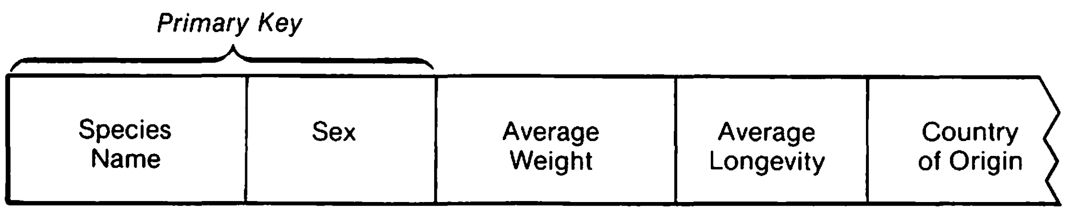


Figure 3-4. Indexed File Record Showing Primary Key Field

The primary key may be any field in the data record. Because DMS must retrieve the entire record to access the primary key, the primary key access time for fixed and variable length records is the same regardless of the placement of the key in the record. For compressed records, DMS uncompresses as much of the record as is necessary to read the primary key; therefore, there is a slight advantage in placing the primary key near the beginning of compressed records.

3.4.4 Primary Key Tree Structure

The principal feature of indexed data files is that you can access individual records by the values of key data fields. DMS performs this access using the file's index blocks. Index blocks can reside anywhere in the file and in any sequence; DMS attempts to place index blocks in optimal locations within the file. Index blocks are logically interconnected by pointers to form logical tree structures, with a single index block at the top and one or more levels beneath. Each index block addresses several blocks on the next lower level. The index blocks at the bottom of the tree point to the file's data blocks.

A key path is a subset of an index tree. It consists of the one index block on each level of the index tree that is used to access a particular record.

Primary Index Block Structure

Primary key index blocks are 2K blocks, some or all of which are initially located in the middle of the first extent of the data file. All primary key index blocks begin with a 2-byte block length prefix (BL), which indicates the current length of the block's contents (i.e., the total space occupied by the table entries and the block length indicator). Every primary key index block terminates with a 3-byte key link pointer (KLP), which points to the next primary key index block on the same index tree level.

An index block is a table of entries. Each entry is a pair of items: a primary key value and a block number. Each entry in an index block contains the highest primary key value (PK) stored in a particular data block. Since records are stored in ascending primary key sequence, the highest primary key value in a data block is also the primary key of the last record in the data block. By having available the highest primary key for each block, DMS can derive the range of primary keys within each block.

The PK is paired with the relative block number of its data block. The length of each table entry is the length of PK plus 3 bytes for the block number. Entries are sorted in ascending primary key sequence, not in data block sequence. The structure of primary key index blocks is shown in Figure 3-5.

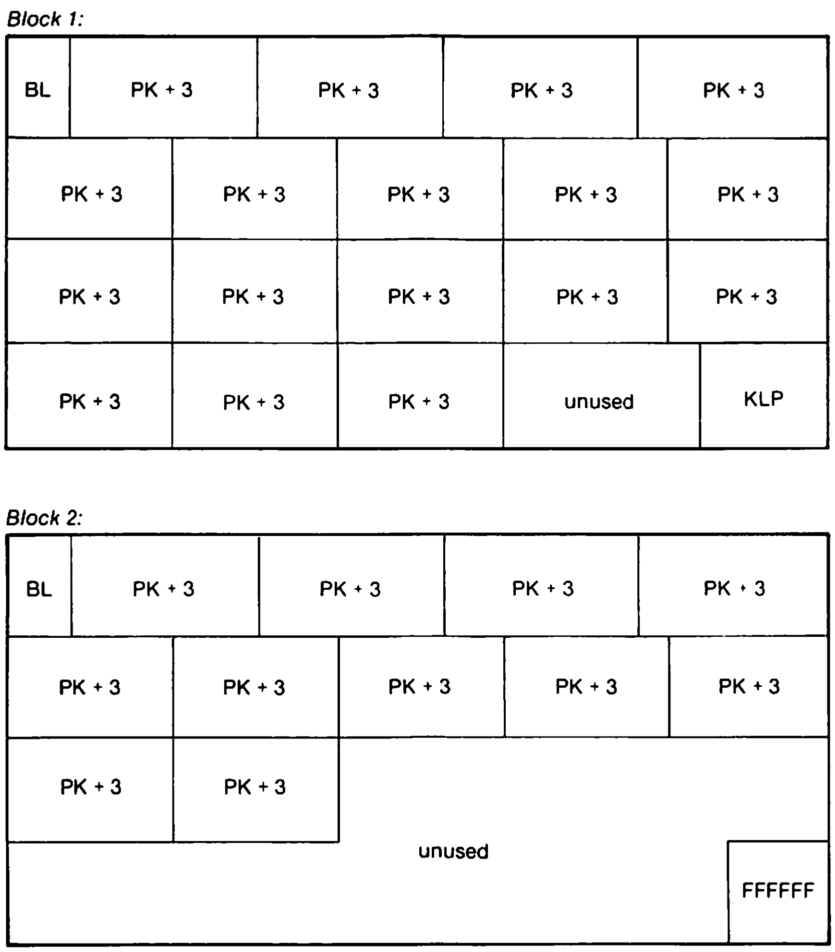


Figure 3-5. Primary Key Index Block Format

Primary Key Index Block Function

In order to read a specific record from an indexed file, DMS searches the index block, using a binary search method, until it locates an entry with a primary key value equal to or greater than the record sought. This entry contains a 3-byte pointer value. DMS follows the pointer to the indicated data block and locates the record sequentially within that block.

For example, to find the data record "Pig" using the index block in Figure 3-6, DMS uses a binary search to locate the index table entry "Rabbit", which is the first entry higher in the ASCII collating sequence than "Pig". DMS then locates block 9, the block pointed to by that entry, and reads the data records in that block until it finds "Pig".

Highest Primary Key Value In Block	*Block Number*
Camel	0
Fox	1
Iguana	2
Lion	3
Opossum	4
Rabbit	9
Skunk	10
Length of uncompressed primary key	*+ 3 bytes*

Figure 3-6. Primary Key Index Table Entries

Root Index Block

In relatively small data files, a single index block is large enough to hold the primary key entry for every data block. In such a case, the primary key index block is also the root index block. However, one index block may not be large enough to store all of the primary key entries. In this case, DMS creates multiple index blocks to store the primary key entries. These multiple blocks are called low-level index blocks. DMS constructs an index tree structure of one or more additional levels to access these low-level index blocks. Additional data records may require further levels of index blocks. The highest level block is called the root index block; it is addressed by the FDR1 record in the VTOC. All other blocks are located from the root index block.

The format of the root index block is the same as the format of other index blocks. The root index entry values consist of the highest value from each of the index blocks on the next lower level, paired with the number of that index block. The low-level index block entry values consist of the highest value from each of the data blocks. A file with two levels of index blocks is shown in Figure 3-7.

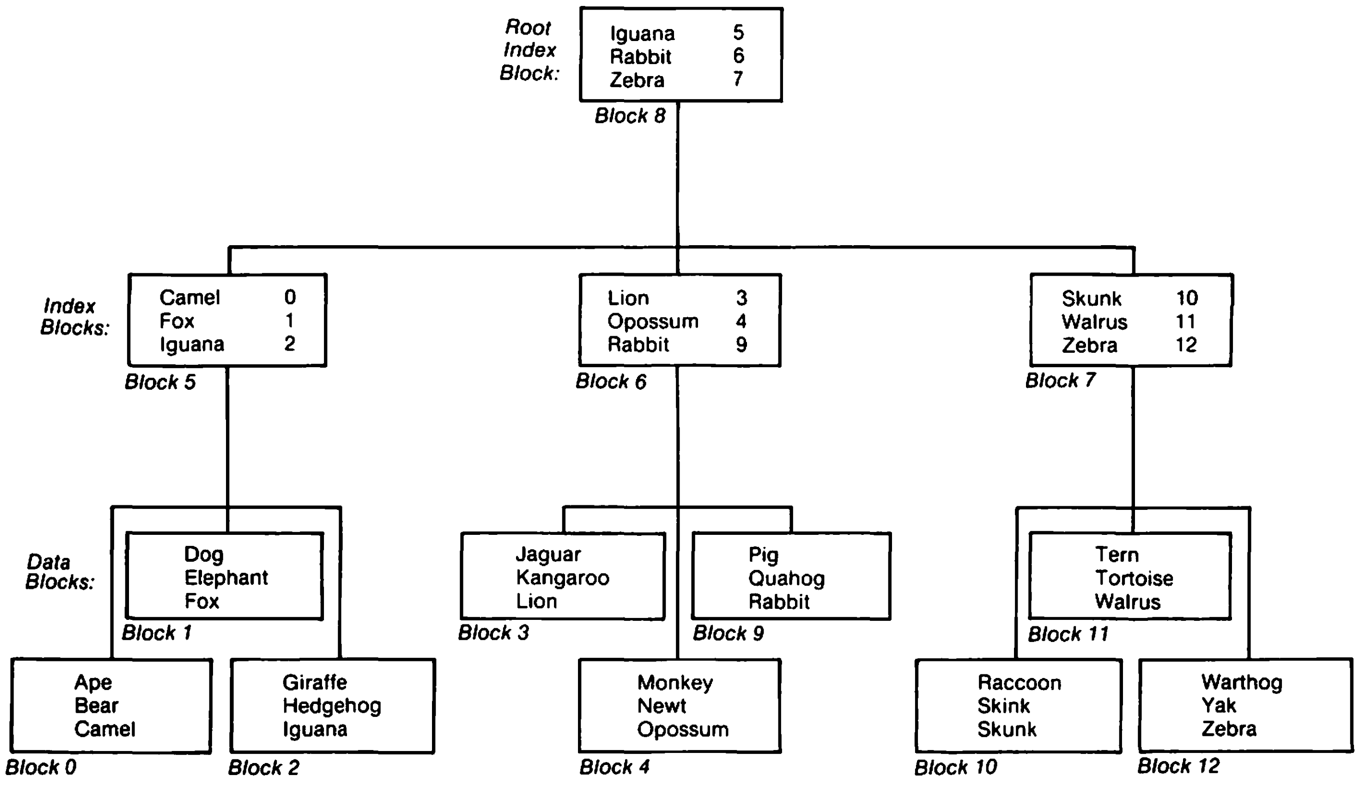


Figure 3-7. Indexed File Tree Structure

The Right-Hand Edge

In an actual file each index block would hold many more than the three entries shown in Figure 3-7. The index block tree shown in Figure 3-7 is also simplified in that each index level contains the value Zebra. On each level, Zebra is the highest-value entry in the last block of that level: the "right-hand edge" of each level of the tree. In an actual index tree (primary or alternate), the field that takes the value Zebra here would instead take high values (represented in hexadecimal as FF). The number of bytes of hexadecimal FFs corresponds to the length of the primary key field.

The reason for this convention is as follows. If Zebra were the actual right-hand index field value, then adding a higher-value data record, Zebu for example, would require rewriting fields on all of the levels of the index tree. With the right-hand edge field set to high values, record access by means of the index tree is unaffected, while file update is simplified. With a right-hand edge of high values, the upper index blocks do not have to be rewritten every time a high-order data record is added or deleted.

3.4.5 The Initial Placement of Blocks in an Indexed File

When you create a file, you estimate the number of data records to be initially put into the file. DMS uses the uncompressed maximum record length and this estimated number of records to calculate the length of the primary allocation. If during indexed file creation DMS finds the estimate to be too small, it closes the file with a file status '24'. DMS does not allocate additional extents for data records to an indexed file in Output mode; however, DMS can allocate additional extents in Output mode to an alternate indexed file for the storage of alternate key index blocks.

DMS establishes a pointer to the last block used in the file, known as the E-Block. The E-Block number is counted from zero. The total number of blocks in the file is represented by the N-Block number. Because this number is counted from one, the E-Block number is always at least one less than the N-Block number. You can display the file length and the number of available blocks at the end of the file using the Manage Files and Libraries option of the Command Processor screen.

The numbering of the blocks in Figure 3-7 demonstrates how DMS builds an indexed file. The index blocks are initially located in the middle of the primary allocation, with the low-level index blocks presented first, followed by each higher level of index blocks in ascending order. The root index block is the last index block built during file creation. If, as in Figure 3-7, data blocks require more than half of the primary allocation, DMS places the remaining data blocks in the file after the index blocks.

If you overestimate the number of records to be written to the file, blank blocks may appear embedded in the file or at the end of the file, as shown in Figure 3-8. If you severely overestimate the number of records, there may not be enough data records to fill the blocks before the first index block, resulting in embedded blocks. You can release unused space at the end of an indexed file at the conclusion of output processing, if the file is not an alternate indexed file. DMS chains together embedded unused blocks, which it uses for subsequent updates to the file.

Underestimating the number of records to be placed in an indexed file results in an error message (File Status '24') indicating that the primary allocation has been exceeded and that DMS has prematurely closed the file before all the records were written to the output file. You must reopen a prematurely closed file in I/O mode to continue writing records to the file. An accurate estimation of the number of records results in the smallest and most efficient data file.

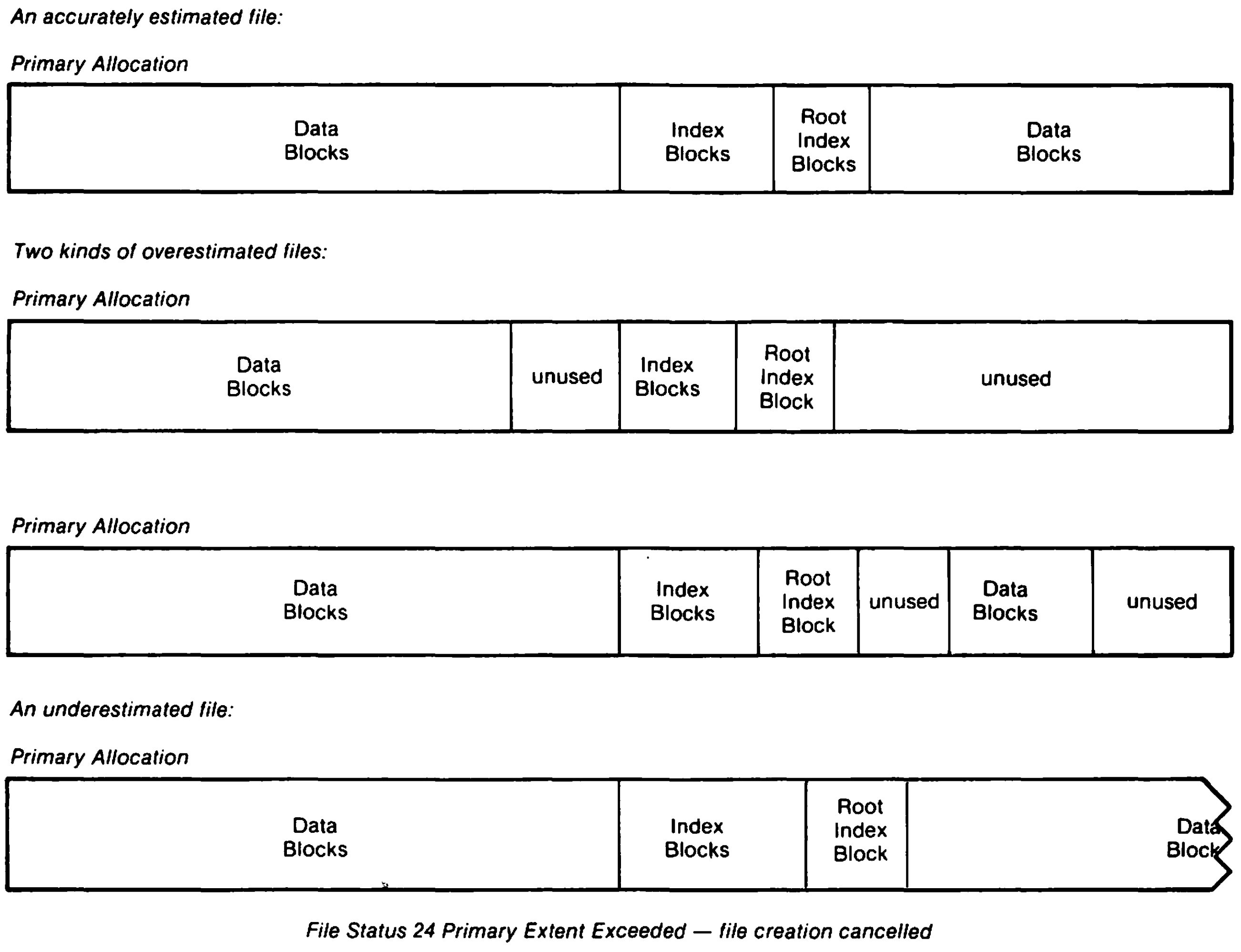


Figure 3-8. Block Locations in Indexed Files

When DMS creates an indexed file, it chains together all embedded empty blocks, so that DMS can locate these blocks when they are needed to enlarge the file. Both empty blocks at the end of the file and embedded empty blocks are available to the file for future assignment as data or index blocks. DMS uses embedded empty blocks in preference to empty blocks located after the E-Block at the end of the file. During file creation, you can either retain or release empty blocks located after the E-Block. However, you must reorganize the file (using the COPY utility) to release embedded empty blocks.

3.4.6 Adding Records to an Indexed File

You can modify the data in indexed files by modifying existing records, adding records, or deleting records.

DMS adds records to a file in the sequence dictated by each record's primary key value. Therefore, you cannot change the value of a record's primary key. To change a primary key value, you must delete the original record and create a new record.

When you add a record to an indexed file, DMS locates space for the additional record by performing the following sequence of steps:

1. It uses available space within the appropriate data block. This free space in the block may have been created when you deleted a record from the file, or may have been set aside when you established a packing density of less than 100%.

2. If insufficient space is available in the block, DMS uses a technique called block splitting. Block splitting places records in an empty block and establishes pointers to enable random and consecutive access of those records. To locate an empty block, DMS first checks the pointer to the chain of embedded blank blocks in the file. If a chain exists, DMS uses the head block of the chain and rewrites the head-of-chain pointer.

3. If no chain of embedded blocks exists, DMS uses an available block at the end of the file, located between the E-Block and the N-Block.

4. If no blocks are available at the end of the file, DMS automatically allocates an additional extent and changes the N-Block number, thus adding the additional extent's blocks to the end of the file as available blocks.

DMS locates space for record storage automatically; all parts of this process are transparent to the user. You may add records anywhere in the file, including records with primary key values less than or greater than those of all of the existing records in the file. To add records to an indexed file, you open the file in I/O mode or Shared mode. Records written in these modes do not need to be in primary key sequence. DMS automatically places each record in ascending ASCII collating sequence by primary key as the program writes it to the file. DMS updates index blocks as each record is written, and readdresses pointers, and adds new index levels as needed.

Once DMS has created an indexed file, it can allocate additional extents until the file reaches a total of thirteen extents. The length of each secondary extent is half the length of the primary extent or smaller. If no free extent as large as half the length of the primary extent is available, DMS allocates the largest available extent.

3.4.7 Block Splitting

A block split divides the contents of a block into two separate blocks. When you attempt to add a record to a file, DMS automatically performs a block split when the block does not have room for the insertion of the additional record. After splitting a block, DMS rewrites pointers to permit records to be accessed in sequence by primary key value.

DMS splits both data blocks and primary and alternate index blocks as needed. DMS performs all block splits in the same fashion, but for the sake of clarity, this manual describes data block splitting and index block splitting separately.

Data Block Splitting

DMS can insert records into the data blocks of an indexed file. This is one of the principal advantages of indexed files over consecutive files. DMS uses one of two methods for updating files by inserting new records:

1. DMS can insert a record into available space in its intended data block. You can establish these spaces at the time of file creation by setting the packing density to less than 100% (see Chapter 9).

2. If a data block contains insufficient empty space for a simple insertion, DMS creates space for the insertion of a record. DMS does this by block splitting, that is, dividing the contents of a block between two blocks.

For example, the zookeeper wants to add the record "Koala" to the data file. As shown in Figure 3-9, the data block containing that portion of the alphabet, Block 3, is already filled (in this example, no block can contain more than three records). To add the new record, DMS must allocate more space than currently exists in Block 3.

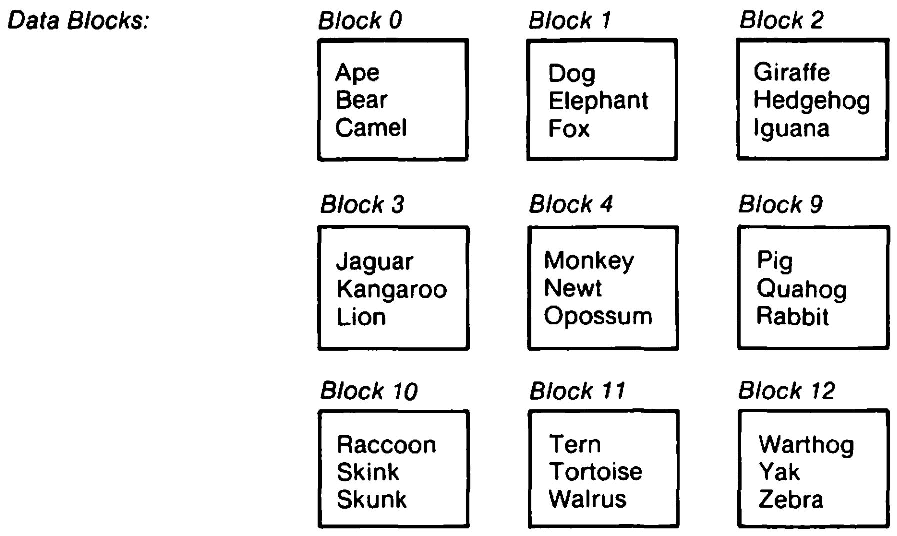


Figure 3-9. Indexed File Data Blocks Prior to Block Splitting

To add the new record, DMS must split Block 3. DMS divides the records in Block 3 into two equal halves; half of the records are retained in the original block, and the other half are placed in an empty block, usually located at the end of the file. The two data blocks created by the block split, Blocks 3 and 13, are shown in Figure 3-10. DMS rewrites the data link pointer (DLP) fields at the ends of Blocks 3 and 13, so that a sequential read of the file reads the blocks in primary key sequence: ...2, 3, 13, 4, 9, 10...

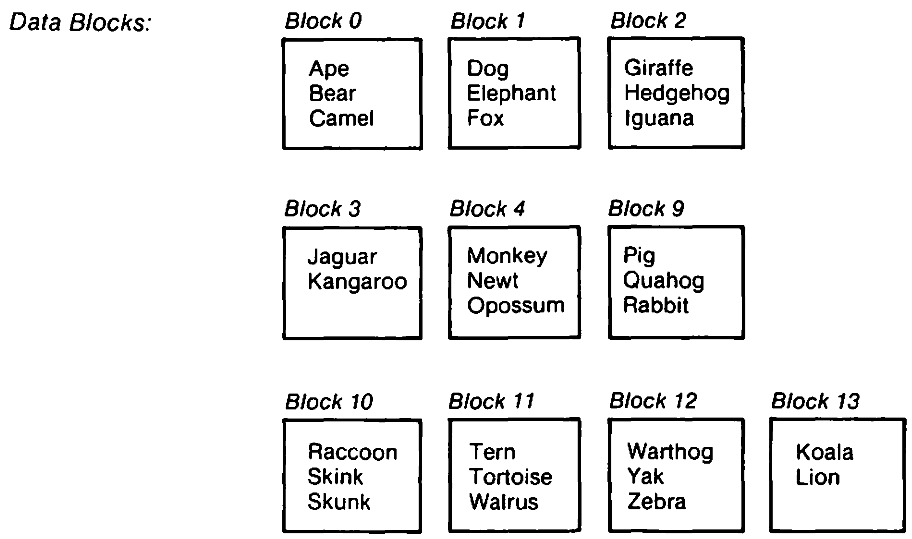


Figure 3-10. Indexed File Data Blocks after Block Splitting

After the block is split, the new record is added to either the original block or the new block, depending on its place in the sequence of primary keys.

Index Block Splitting

Changes made to the data blocks often result in changes to the index blocks. Since the low-level index blocks contain a table entry for each data block, the splitting of a data block necessitates updating one or more index blocks by adding a table entry. Splitting a data block may require the splitting of an index block. Figure 3-10 shows the zookeeper's data blocks after a data block split; this data block split necessitates a block split to the file's index blocks, shown in Figure 3-11. DMS must now represent Data Blocks 3 and 13 with entries in Index Block 6.

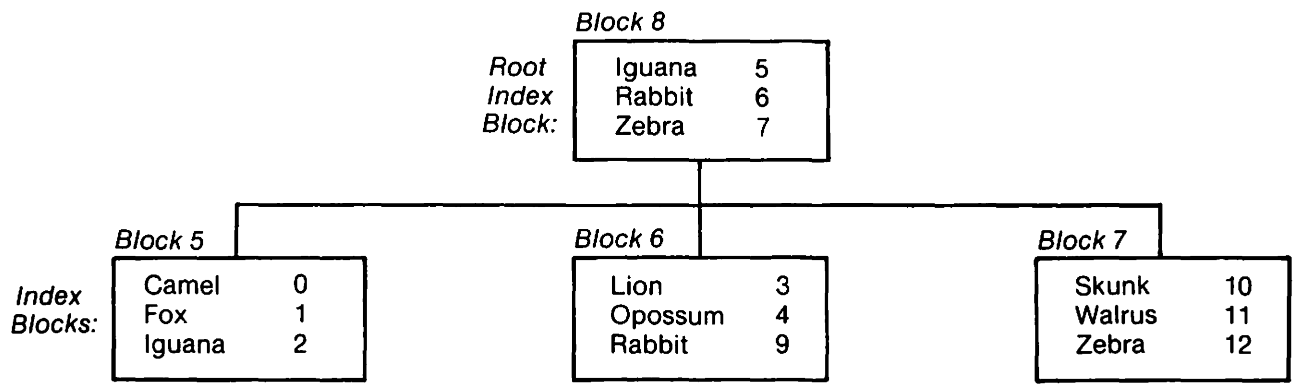


Figure 3-11. Index Tree Structure before Block Splitting

In the case shown in Figure 3-11, there is no room to include additional entries in any of the index blocks, so a data block split necessitates an index block split on each level of the index tree, and the creation of a new root index block level. Figure 3-12 shows the file's index trees following index block splitting.

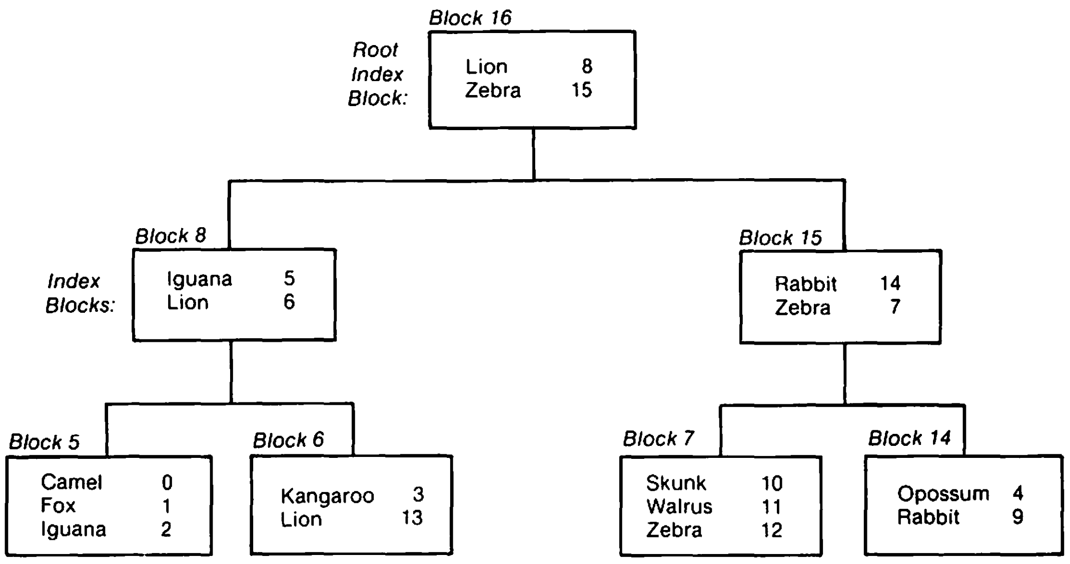


Figure 3-12. Index Tree Structure after Block Splitting

In Figure 3-12, the contents of Block 6 have been split between 6 and 14; the contents of Block 8 have been split between 8 and 15, and a new root index block, Block 16, has been created.

The key link pointer (KLP) at the end of an index block chaining it to the next block on the same index tree level must be rewritten if the index block has been modified. DMS rewrites the KLP of Block 6 to point to Block 14, and the KLP of Block 14 to point to Block 7. It then rewrites the KLP of Block 8 to point to Block 15, and the KLP fields of Blocks 15 and 16 to take high values (hexadecimal FFFF).

Requiring the creation of four new blocks (as in the above example) when adding the first new record, and rewriting several address fields is time-consuming. You can minimize this overhead by setting packing densities to less than 100%. Packing densities are described in Chapter 9.

You cannot access a file that has undergone numerous block splits as rapidly as a new file, because DMS must skip to the end of the file to read split blocks. You can, however, reorganize a file to minimize skipping by running COPY with REORG. This utility rebuilds the index tree structure, and reassigns records to blocks based on their primary key values.

3.4.8 Deleting a Record from an Indexed File

When you delete a record from an indexed file, DMS automatically shifts the records in the block in order to occupy the space left between records by the deletion. DMS decreases the value of the block length indicator to reflect this shift. If the deleted record contains the highest primary key value in the block, DMS modifies the index block entry for that block by copying the next-highest primary key value from the data block.

DMS does not automatically reorganize index blocks to balance index trees. If you have performed extensive deletion on a file, the index tree can contain more blocks and levels than are required, decreasing performance efficiency. Running COPY with REORG restructures the tree to reflect the actual number of record blocks.

3.5 ALTERNATE INDEXED FILES

In addition to data access by means of a primary key, DMS supports file access using up to 16 alternate keys. Alternate index keys provide alternate paths for rapid access to individual records. For example, our zookeeper uses the name of each species of animal as a unique primary key. However, cases frequently arise that require access of animals' records by country of origin. Making the country of origin an alternate key would improve file usability.

An alternate indexed file contains all of the primary key index blocks that a regular indexed file contains. All sequential and indexed functions available for indexed files are also available for alternate indexed files. In addition, DMS builds a separate index tree for each alternate key defined for the file.

You should exercise restraint in creating additional alternate keys, because the creation of each alternate key results in the creation of a separate index block tree structure. You should only use alternate keys for accessing single records in a file. If batch reporting of multiple records is required, such as a report of all employees alphabetized by last name, it is much more efficient to sort the file by last name (using the SORT utility) than to access a large number of records by an alternate key.

3.5.1 Alternate Keys

An alternate key can be any field or contiguous group of fields in a record. An alternate key can be up to 254 characters in length; however, the use of short alternate keys is strongly recommended, as it greatly facilitates efficient file processing. The length of the primary key limits the maximum length of an alternate key; the total of the primary key length and the longest alternate key length cannot exceed 255 characters.

Minimizing the length of alternate keys is an important way to conserve storage space in an alternate indexed file. When you establish access keys, avoid making the alternate key any longer than is necessary to provide unique and easily used values. For example, the first five or six letters are probably sufficient for a last name field alternate key.

You can establish more than one alternate key for a data file. However, although 16 alternate keys are available, you should not create a file with any more alternate key paths than are absolutely necessary.

Unlike a primary key, an alternate key can be assigned non-unique values. For example, the last name field of an employee file would in most cases not be a good choice for a primary key field; last names are not sufficiently unique to ensure there would never be more than one person with the same last name. However, the last name field can be used as an alternate key. You must simply specify whether or not duplicate alternate key values are to be allowed for each alternate key when the file is created.

Every record in an indexed file (primary or alternate) must contain a primary key value. However, every record need not contain a value for each alternate key field; only those records to be accessed by a particular alternate key must contain a value for that alternate key field. Records with an alternate key field can be established as accessible by that key field, or as not accessible by that field. For each record, you decide which alternate key fields are to be used as keys when you write or rewrite the record.

Primary and alternate keys are usually separated fields in a data record, but they need not be. Figure 3-13 illustrates the possible relations between primary and alternate key, or between two alternate keys.

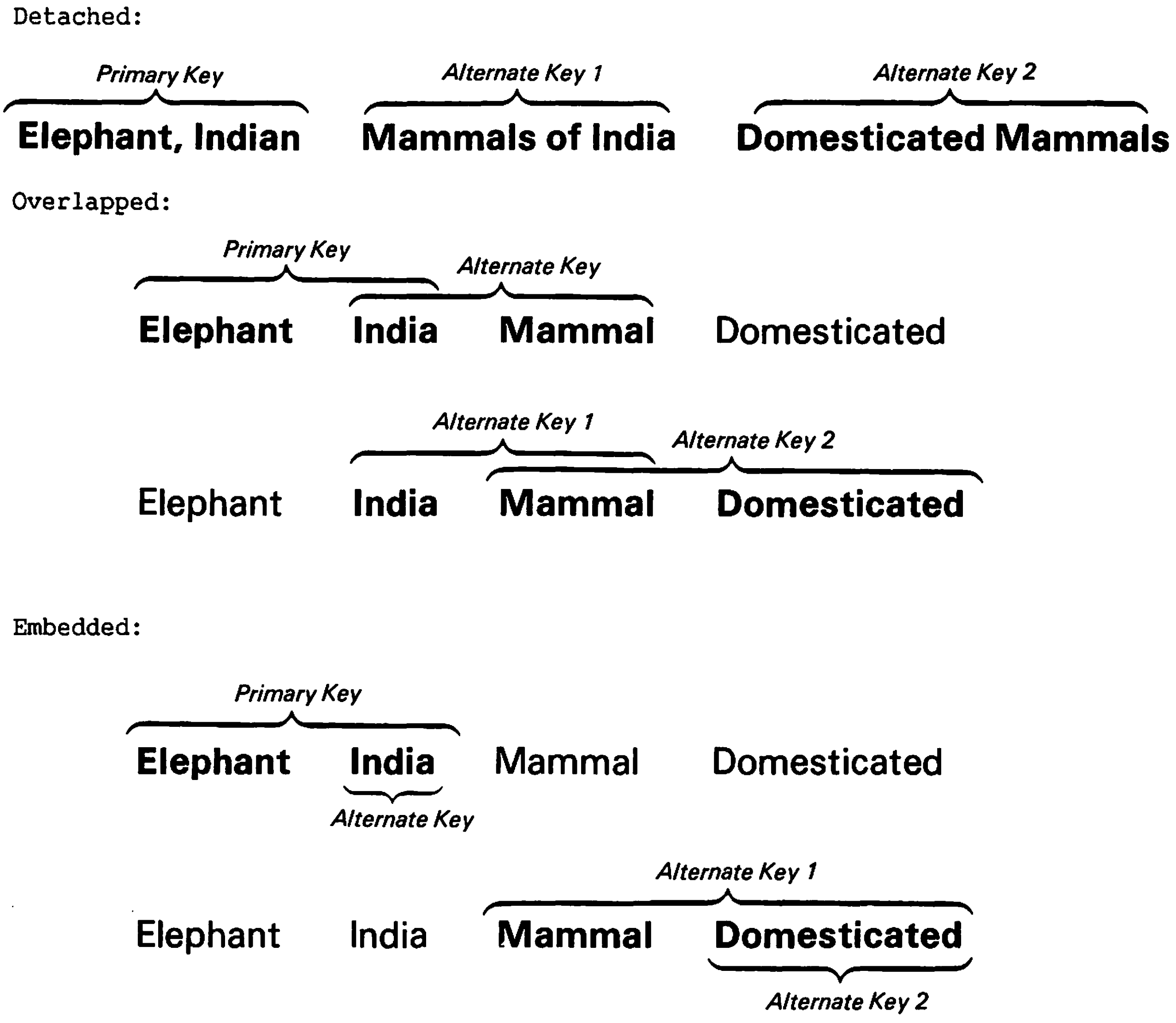


Figure 3-13. Relations between Key Fields in a Data Record

Overlapping or embedded an alternate key and a primary key can affect the space requirements for the alternate index tree structures. The overlap of the two keys reduces the total block space required for each alternate index key entry. See Chapter 15 for details. The overlapping or embedding of two alternate keys has no performance impact on DMS.

3.5.2 Alternate Key Tree Structure

DMS uses two different sets of index blocks in forming alternate key index tree structures: one set for alternate keys that take only unique values, and another for alternate keys that can take duplicate values.

NOTE

Data files created prior to Operating System Release 5.3 use the tree structure for unique values for all alternate indexed files. These files will continue to be supported. However, you can improve efficiency by converting old files that contain duplicate values to the new tree structure. You can convert a file by copying it using the COPY utility with REORG=YES.

Pseudo-record Index Blocks for Alternate Indexed Files

To access a file by an alternate key, DMS uses a tree of alternate index blocks. An alternate index tree always contains a single alternate index root block at the top of the tree. Each entry in the root block points to an alternate index block at the next lower level. The lowest-level alternate index blocks are pseudo-record blocks. Their entries point directly to the primary index tree. Given a record's primary key, DMS can use the primary index tree to locate the individual data record by primary key value.

Pseudo-record index blocks are organized as conversion tables from alternate to primary keys. Each pseudo-record block contains paired entries for each data record, matching the alternate key value to that record's primary key value. Each pair of key values constitutes a single pseudo-record. Pseudo-record blocks can be of two types, those that contain duplicate alternate key values, and those that contain unique alternate key values.

The example of the zookeeper's data file shows how an alternate indexed file is structured. Two alternate keys have been established for the zookeeper's data file. The scientific name of each species constitutes a unique alternate key; the country of origin of the species is an alternate key that permits duplicates. The structures of these two types of pseudo-record index blocks are the same, as shown in Figure 3-14.

Pseudo-record Index Block with Unique Alternate Key Values:

Alternate Key/Primary Key
Elephas/Elephant
Ericius/Hedgehog
Leo/Lion
Testudo/Tortoise
Ursus/Bear
Vulpes/Fox

Pseudo-record Index Block with Duplicate Alternate Key Values:

Alternate Key/Primary Key
Africa/Elephant
Africa/Lion
Alaska/Bear
England/Fox
England/Hedgehog
Galapagos/Tortoise

Figure 3-14. Alternate Index Pseudo-record Block Types

DMS sorts pseudo-records by their alternate key values. If duplicate alternate key values exist, DMS sorts the pseudo-records by primary key value within that alternate key value, as shown in Figure 3-14.

Since DMS creates one pseudo-record for each data record on an alternate key path, in most cases there are more pseudo-records for an alternate key path than can be fit in a single block. If any record are placed on an alternate key path, DMS builds a tree of alternate index blocks, surmounted by a single root index block for that alternate key path. The minimum tree size is two blocks: a root index block and a pseudo-record block.

The upper levels of the alternate index tree can have two different block structures, one for alternate keys that take only unique values and another for alternate keys that allow duplicate values. You must specify, when you create a file, whether or not duplicate alternate key values are permitted for a particular index path. Multiple alternate key paths are independent of each other; you can create unique and non-unique alternate keys fields on the same data record.

Regardless of type, all alternate index tree blocks, like all primary index tree blocks, begin with a 2-byte block length indicator, and end with a 3-byte key link chain containing the address of the next index block on that level of the tree.

Alternate Index Tree Structure for Unique Alternate Key Values

For unique alternate key values, the alternate index tree levels above the low-level, pseudo-record blocks are in the same format as the upper levels of a primary index tree. That is, for each entry, DMS pairs the highest key value in a lower-level block with the relative block number of that lower-level block. DMS can create up to fifteen levels of these kinds of blocks. The entries at the lowest of those levels point to pseudo-record blocks. DMS searches the tree until it locates a pseudo-record index block. Each pseudo-record block contains pairs of alternate and primary key values, providing the primary key value for searching the primary index tree. (See Figure 3-15.)

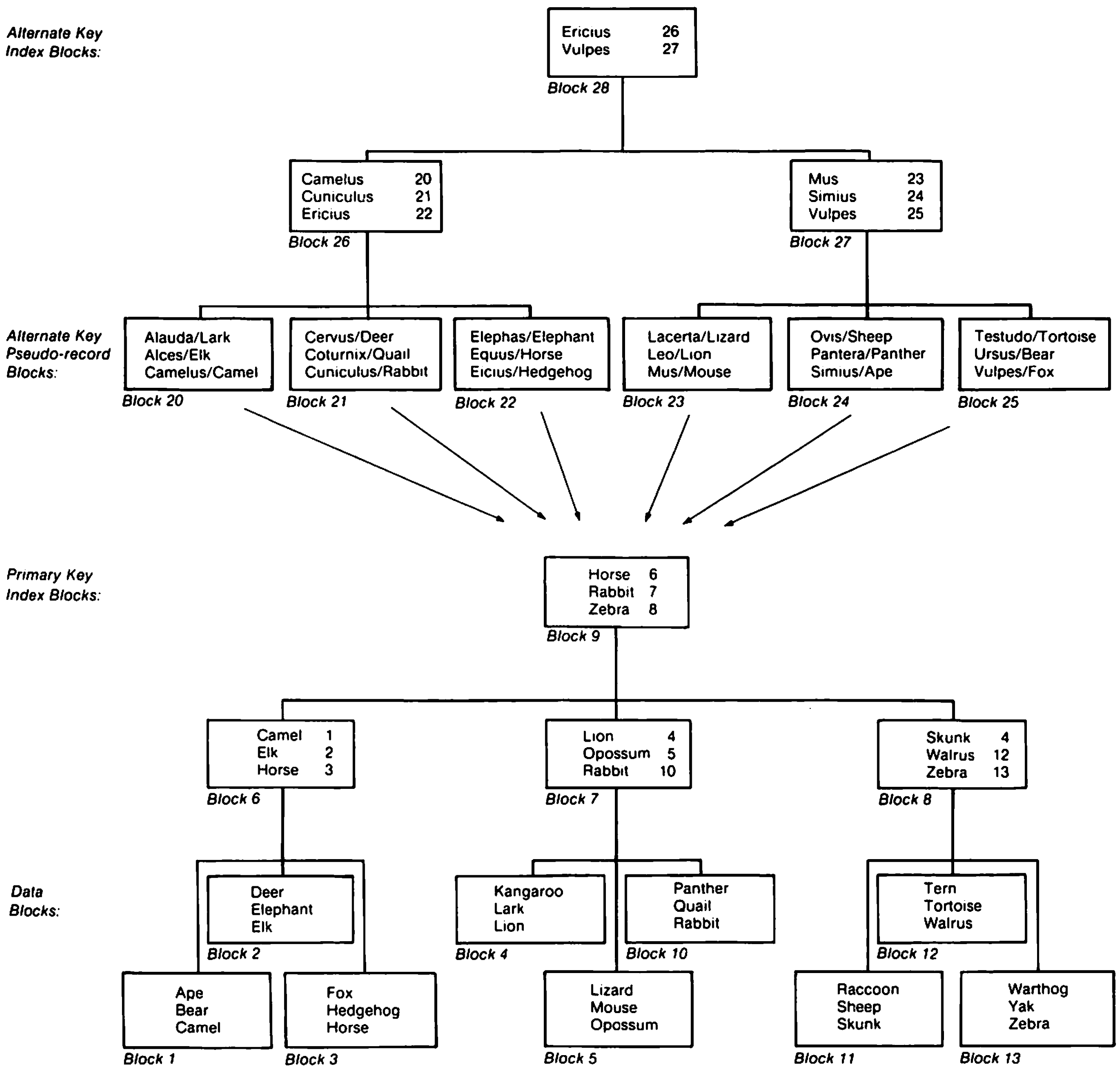


Figure 3-15. Tree Structure for an Alternate Key Path with Unique Values

Alternate Index Tree Structure for Duplicate Alternate Key Values

When you initially create an alternate key, you can specify whether two or more records using that key field can have the same alternate key value. If you permit duplicate alternate key values, DMS creates a different block structure for all of that key's higher-level alternate index blocks. These upper-level alternate index blocks are similar to the lower-level alternate index pseudo-record blocks shown in Figure 3-15. Alternate index blocks for keys with duplicate values contain paired alternate and primary key values on all levels. In addition, the higher-level alternate index block pseudo-records contain the block number of the block on the next lower level. An alternate index tree for a key permitting duplicate values is shown in Figure 3-16.

DMS facilitates access to the appropriate duplicate alternate key by pairing the unique primary key value with the alternate key value at every level of the tree structure. In this way, DMS accelerates access to a record whose alternate key has many duplicates values, such as the value "Africa" in Figure 3-16. Sequential reading through alternate index blocks is minimized by the concatenation of alternate and primary key values, giving each record, in effect, a unique alternate key value.

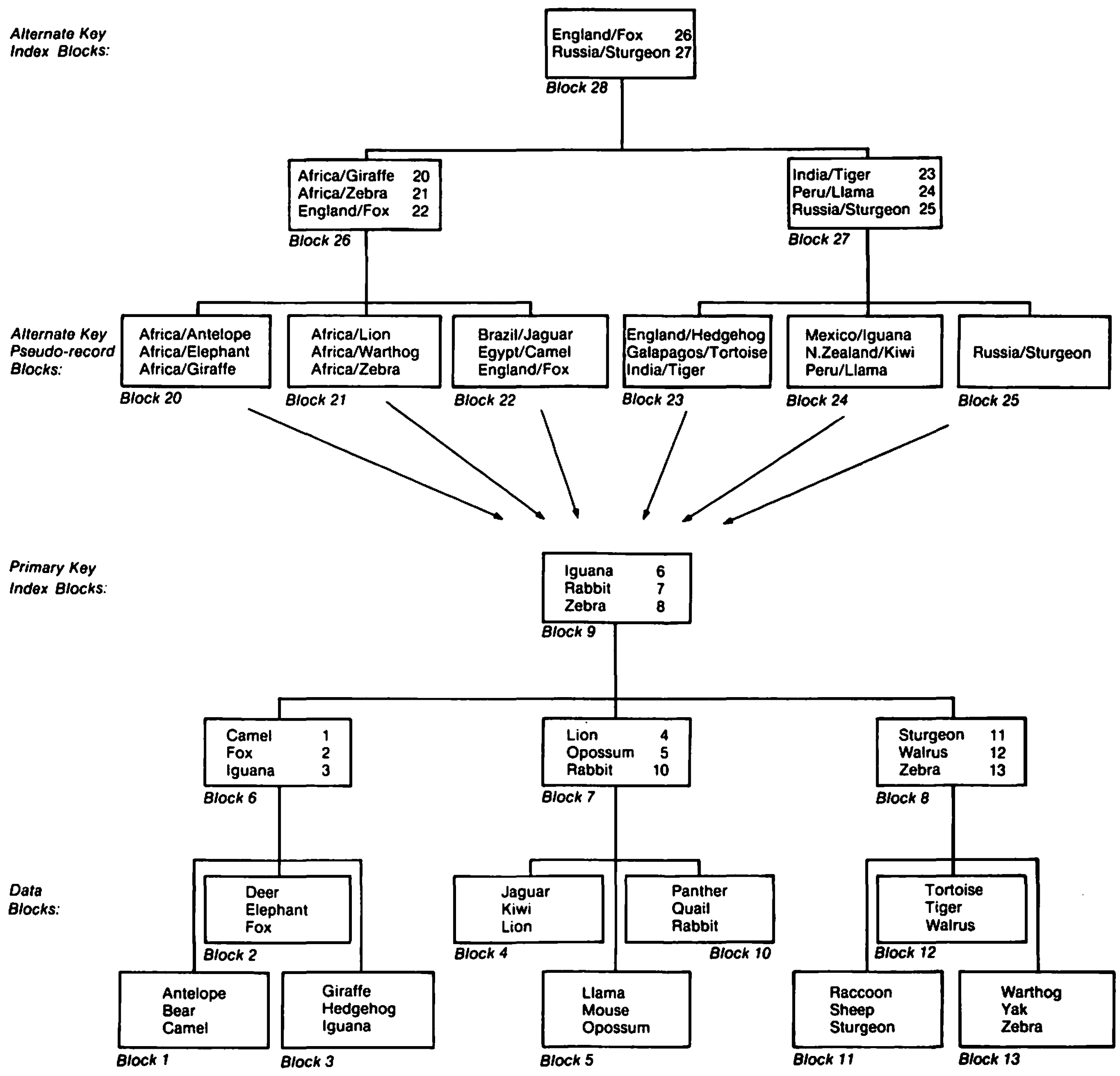


Figure 3-16. Tree Structure for an Alternate Key Path with Duplicate Values

Because the entries for the higher index blocks are longer for duplicate alternate keys than for unique alternate keys, you can minimize both space and access time by using unique alternate key values wherever possible. For example, an alternate key could be established on our zoological file to separate aquatic from land animals. But such an alternate key would contain mostly duplicate values. A more efficient method of access would either be to include an aquatic/land indicator as part of the primary key field, or to use the SORT utility to generate species list files for the two habitats.

3.5.3 The AXD1 Block

Every alternate indexed data file contains an Alternate Index Descriptor (AXD1) block, which is always stored in logical block number zero of the alternate indexed file. This block contains control information for all the alternate key paths in the file. This information includes a bit mask (the PMASK) indicating the alternate index paths that are defined for the file, the ALTINX byte indicating the alternate index path currently in use, pointers to the the root index block of each of the alternate key trees, and information specific to individual alternate key paths. You can establish up to sixteen alternate keys, each with its own alternate index tree, in the AXD1. An indicator for each alternate key path specifies whether records containing only unique alternate key values, or those containing duplicate alternate key values can be written to the file.

Each record in an alternate indexed file contains a sixteen-bit field called the bit mask suffix. Prior to writing a record to an alternate indexed file, you write this suffix field value to the sixteen-bit AXD1MASK parameter in the AXD1. It is only necessary to write this AXD1MASK parameter when you create a record or change the alternate key access of an existing record.

Two sixteen-bit mask fields are located in the AXD1, the PMASK and MASK fields. The PMASK, or primary mask field, contains an ON bit for every alternate key established for the file. PMASK is a read-only field, established and maintained by the system; do not modify the PMASK value. During record processing the PMASK does not change. The MASK field holds the current record's 16-bit suffix mask. Each ON bit of the MASK represents an alternate key available to that record. When you read a record, DMS places the record's suffix mask value in the AXD1 MASK field. When you write or rewrite a record, you supply a suffix mask value to the MASK field. DMS then checks the MASK field value against the PMASK field. The MASK field value must be a logical subset of the PMASK field; that is, each bit that is set ON in the MASK field must also be turned ON in the PMASK field. See Figure 3-17. After checking your MASK field value against the PMASK field value, DMS writes the record to the file, copying the value of the MASK field into the record's 16-bit suffix mask.

PMASK field:

1110 0000	0000 0000

2-byte alt. key bit mask

A file with three alternate key paths.

MASK field:

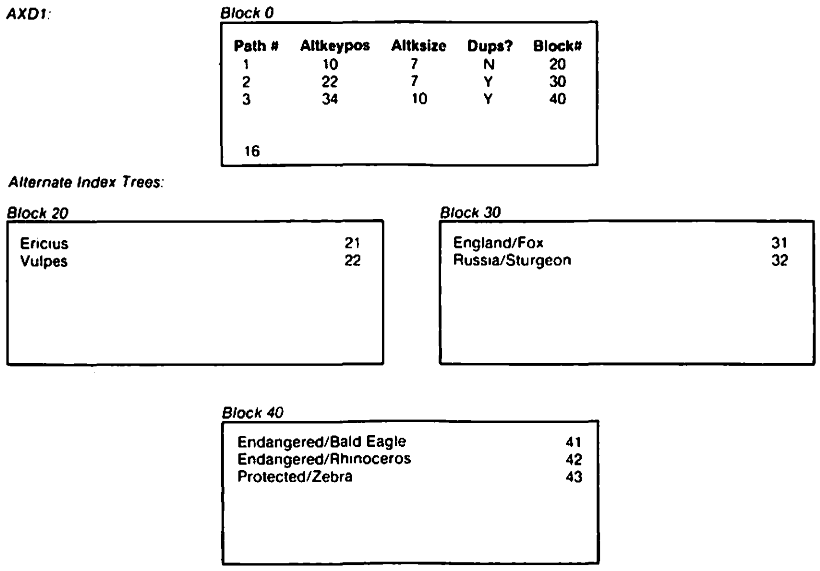
1010 0000	0000 0000

2-byte alt. key path bit mask

A record accessible by two alternate key paths: alternate keys 1 and 3.

Figure 3-17. Mask Fields within the AXD1 Block

In addition to specifying how many alternate keys are in the file, the AXD1 specifies the alternate index path currently in use, the length and location of each alternate key field, the type of values acceptable (unique or duplicate), and the relative block number of the root block of the alternate index tree. See Figure 3-18.



AXD1:

Block 0

Path #	Altkeypos	Altksize	Dups?	Block#
1	10	7	N	20
2	22	7	Y	30
3	34	10	Y	40
16				

Alternate Index Trees:

Block 20

Ericius	21
Vulpes	22

Block 30

England/Fox	31
Russia/Sturgeon	32

Block 40

Endangered/Bald Eagle	41
Endangered/Rhinoceros	42
Protected/Zebra	43

Figure 3-18. Schematic of AXD1 and Alternate Index Root Blocks

See Chapter 6 for details on generating an AXD1. Refer to the READ KEYED and START EQ function requests in Chapter 7 for information on retrieval by alternate keys using Assembly language. For higher-level language access, see Chapter 5 and the individual language reference manuals.

3.5.4 Records on an Alternate Key -- Selective Indices

Rationale

Not all records have to be accessible by each alternate key path. In fact, DMS initially assigns space to alternate indexed files on the assumption that only one half of the records in the file will be on each alternate key path. For example, the third alternate key shown in the Figure 3-18, the Endangered Status key, is a field placed in the records of all species that are endangered, protected, or extinct. Records of species that do not fall into any of these categories need never be accessed using this alternate key. The Records-on-Key feature prevents DMS from building pseudo-records for these never accessed records. Never accessed records do not need to occupy space in the alternate index tree for that key path.

Mechanism

DMS supplies each record in an alternate indexed file with a two-byte suffix denoting which keys apply to which alternate index key path. These two bytes function as a bit mask, with a bit set ON or OFF for each of the sixteen possible alternate index key paths. These two bytes are included in the record length prefix, but not in the RECSIZE. The bit mask is never compressed.

The alternate indexed record shown in Figure 3-19 is accessible by alternate index keys 1 and 3. It is not accessible by alternate key 2, or by alternate keys 4 through 16, if indeed these alternate keys exist. The actual number of alternate index keys available to a file cannot be deduced from a record bit mask.

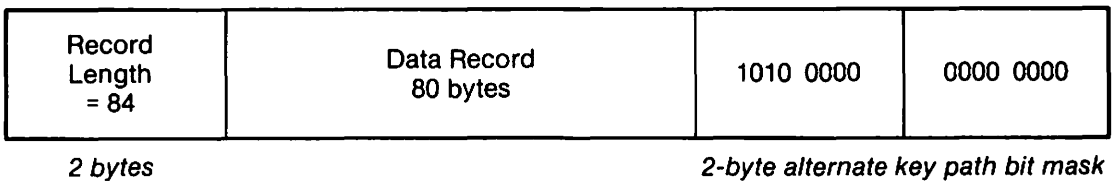


Figure 3-19. Alternate Indexed Record Showing Bit Mask Suffix

In Assembly language you must create the appropriate bit mask for each record in an alternate indexed file. In high level languages the system creates the record bit masks, as described in Chapter 5.

When DMS adds a record to an alternate indexed file, it first matches the record bit mask to the permanent mask (PMASK) field in the AXD1. If the bits in the record mask correspond to the PMASK bits, a DMS creates a pseudo-record for each ON bit in the record mask. If the record mask bits do not correspond to the PMASK bits, the record is not added to the file. See Figure 3-20.

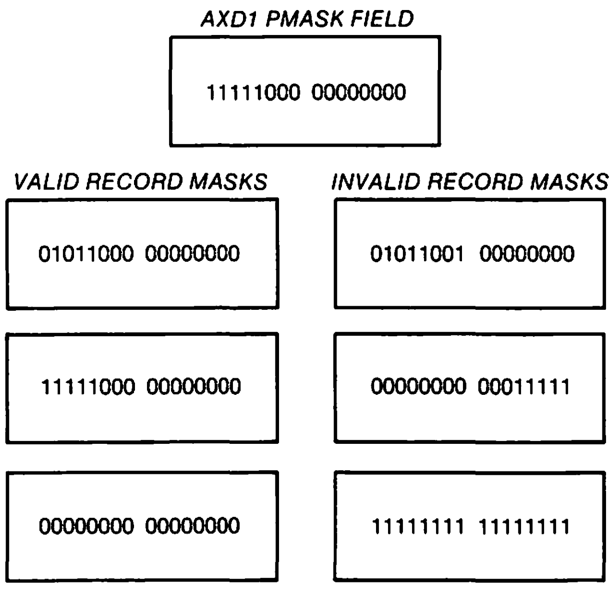


Figure 3-20. Valid and Invalid Record Mask Suffixes for an AXD1 PMASK

3.5.5 Overhead of Multiple Alternate Keys

When a file is created with alternate keys, DMS assumes that 50% of all the records in that file are on each of the alternate key paths. It allocates space in the file based on the assumption that the length of all the alternate keys fields are equal to the maximum uncompressed length of the longest alternate key on any path. Therefore, DMS calculates the size of the primary allocation to be larger than needed if you place fewer than 50% of the records on each key path, or if alternate keys vary widely in size. The primary allocation is smaller than needed if you place more than 50% of the records on each key path. You should bear this in mind when estimating the number of records in the file, and deliberately overestimate or underestimate the number of records in the file slightly to compensate for the number of records on alternate key paths. See Chapter 15 for further details on the creation of alternate index trees.

To add an additional alternate key to an existing alternate indexed file you must use the CREATE user aid or a user-written program to process every record in the file. This can be very time-consuming for large files. See the VS User Aids Reference for the CREATE user aid. If you anticipate that a field may be useful as an alternate key, you should create that alternate key in the AXD1 at file creation time, and put no records on the alternate index key path. Initially, DMS reserves the requisite number of pseudo-record blocks for the unused path, but since these blocks are unused, they are available to be used for block splitting or other operations during normal processing. The AXD1 logic for the alternate key remains, and when that key becomes necessary, you can assign records to it without changing the file structure.

PART II
Data File Access

CHAPTER 4
AN OVERVIEW OF ACCESS FUNCTIONS

4.1 INTRODUCTION TO FILE ACCESS

To access a DMS file from a program, you must specify file definition parameters. When you compile (or assemble) the program, the compiler (or assembler) uses these parameter values to fill in the fields of a User File Block (UFB). DMS then uses the parameter field values in the UFB to locate a pre-existing file or create a new file.

When you compile a high-level language program, the complier builds a separate UFB in user Segment 2 for each file opened by the program. When you assemble an Assembly language program, you specify a UFBGEN for each UFB you wish to build in user Segment 2. UFBs are maintained for the duration of the program run. The first time you open a file, DMS copies the permanent attributes of the file from the UFB into the disk Volume Table of Contents (VTOC). Dynamic attributes of the file, such as the number of records in the file and various pointer values, are copied from the UFB into the VTOC each time you close the file.

How you supply parameter values to the UFB differs from language to language. The system can write parameter values to the UFB when the program is compiled, or write parameter values dynamically at runtime as part of the Open operation. In either case it writes the parameters to the same fields in the UFB, overwriting any previously established or default value.

4.2 ACCESS METHODS

When opening a file you select an access method, specifying whether data processing will be performed in physical units (blocks) or logical units (records). DMS supports three access methods: Record Access Method (RAM), Block Access Method (BAM), and Physical Access Method (PAM). You can use all three access methods on any type of disk or tape file. The access method you select determines the amount of DMS file support; the program must supply file support not provided by the access method.

RAM is the only access method that processes logical units of data (i.e., user-defined data records). It also provides the most complete DMS support. RAM is the default in access method selection; it is the only access method accessible from higher-level languages -- BAM and PAM are accessible only in Assembly language. Many DMS functions, such as index tree creation and data access by key values, location of data records within a block, file sharing, buffer pooling, and compression are only supported in the Record Access Method. Unless otherwise noted, this manual assumes use of the Record Access Method.

BAM reads and writes disk data in 2K byte block units only. Data transfer and copying in block units is considerably faster than in single record units. DMS maintains system buffering in BAM; it reduces overhead by not performing record blocking and deblocking.

PAM offers the greatest flexibility and least DMS support of the three access methods. It allows you to transfer data in multiples of 2K from 2K to 18K. It also allows you to establish specialized buffering strategies for the application. PAM is recommended primarily when data movement is to be minimized or when a flexible user-supported buffering scheme is desired.

For further information on access methods, refer to Chapter 10.

4.3 OPENING AND CLOSING DATA FILES

In order to create a file or access data in an existing file you must first open that file. In some high-level languages (e.g., RPG II) files are opened automatically, but in most cases you open a file by coding an Open statement. An Open statement makes use of file definition parameters to locate an existing file, or to establish tape header information or allocate disk space to create a new file. Following processing of the file data, you should close all opened files. Closing a file releases all resources, making them available to other users, and updates various control blocks.

Prior to opening a file, you must supply to DMS the parameters that describe the file. These parameters provide information such as the following:

File Identification Parameters

- The names used within the program to refer to the file and its record area.
- The permanent external file, library, and volume names used to locate the file.

File and Record Organization Parameters

- The type of records in the file (fixed length, variable length, or compressed.)
- The type of file (data storage, workstation, program, log, printer, or WP)
- The structure of the file (consecutive, relative, indexed, or alternate indexed.)

Space Allocation Parameters

- Information used to allocate sufficient space for a new disk file (maximum record size and estimated number of records.)

Key Identification Parameters

- The length and starting location of each primary and alternate key.
- How many alternate key paths are to be created, and which alternate keys allow duplicate key values.

Efficiency Option Parameters

- The size and nature of a buffer area, if desired.
- The packing density of data and index blocks, if desired.

You supply other parameters to the Open statement that do not describe the file to be processed, but describe how the processing is to be performed. You can specify whether or not file definition parameter respecification is to be performed from the workstation at runtime. You can also specify the address of error routines in the event of DMS file status errors.

These parameters are stored in a UFB, which is compiled along with the program data areas into the user's Segment 2 area. You need not assign a UFB to a specific data file when it is compiled; you can assign a UFB to a data file interactively when running the program. You can reassign the same UFB to several files, provided that the UFB is assigned to only one open file at a time.

When DMS receives an Open statement, it locates the UFB and reads the needed file definition parameters. Depending on the values you establish for the file definition parameters, Open establishes user access to different function request subroutines for processing that file. Open also restricts other users' access to the file, based upon the mode specified.

4.4 ACCESS MODES AND SHARING

Records in a file are accessed in a particular access mode. The mode limits record access to read only, write only, or update (both reading and writing). You specify the mode either prior to opening the file, or as part of the Open statement. In order to change the mode, you can close the file and reopen it in another mode. In some cases, you can use the Start function request to switch modes without closing and reopening the file. The access mode indicates the kind of I/O access a program may perform on that file, and establishes certain restrictions on file processing. The five DMS access modes are:

Output	Used to create a new file and write data to it. All files are initially created in Output mode. A file is only opened once in Output mode. If you attempt to open an existing consecutive or relative file in Output mode, DMS displays a warning screen. If you continue, it deletes the existing file and creates a new file with the same file name.
Input	Used for read-only access to the data within an existing file. Records read in Input mode cannot be rewritten to the same data file. Many users can have the same disk file open in Input mode simultaneously.
I/O	Used to read, add, delete, or update records in an existing file. Records modified in I/O mode can be rewritten to the file from which they were read.
Extend	Used to write additional records to the end of an existing relative or consecutive file.
Shared	Used to read, add, delete or update records in an existing file, while allowing other users to simultaneously update the same data file. A record read in Shared mode may be held to prevent another user from modifying the same record. Records modified in Shared mode can be rewritten to the file from which they were read. All consecutive files on disk can be opened for shared I/O processing; some consecutive files can also be opened for for shared output processing.

Shared mode is not supported by the Block Access Method (BAM). Neither Shared nor Extend modes are supported by the Physical Access Method (PAM).

Which access mode you select affects simultaneous access to the file by other users. If you open a file in Input mode, other users can concurrently open the file in Input mode for read-only processing. If you open a file in Output, I/O, or Extend mode, no other user can access the file until you close it. These modes give you exclusive rights to read or update the file. The Shared mode gives multiple users concurrent read and update access to the same file.

In DMS Sharing, each user can exclusively hold one resource at a time (a file, a record, or a group of records or files). In order to hold a new resource, you must release the resource currently held. No incremental resource holding is permitted. Further details on file sharing are found in Chapter 8.

With DMS/TX, each user can hold more than one resource, acquiring resources incrementally as needed by the program. DMS/TX allows each user to claim multiple resources as needed and to hold these resources for the duration of a transaction. DMS/TX automatically resolves conflicts between two users both claiming the same resource. For further details on DMS/TX sharing, refer to the VS DMS/TX Reference.

4.5 FUNCTION REQUESTS PROVIDED BY DMS

Once you have opened a file, you access individual records by issuing instructions known as function requests. Each function request specifies the file on which the operation is to be performed. You must open the file in the appropriate mode for successful execution of a function request. Each function request operation manipulates the requested data and performs maintenance operations to insure that control blocks, index trees, pointers and indicators correctly reflect any change to the data.

The successful execution of a function request can modify your currency pointer location in the file. The currency pointer value is recorded in the file's UFB; it specifies the relative record location of the record requested by the last successful function request. DMS uses this information to perform sequential operations within the file. The currency rules for relative files differ from the currency rules for other file types; in relative files only Read and Start function requests reset the currency pointer.

DMS provides five function requests: Read, Write, Rewrite, Delete, and Start. Some function requests take a modifier that further defines how the function request is to be performed. The five function requests are described in the following sections.

4.5.1 The Read Function Request

A Read function request reads one logical record in RAM. The Read function request can locate a record sequentially, by relative record number, or by primary or alternate key value. The Read function request can be used to hold an individual record, preventing other users from accessing that record. In BAM or PAM, a Read function request reads a block or group of blocks of data.

A Read operation is a multi-step process. During a Read operation, DMS searches the buffer for the appropriate record. If it does not find the record in the buffer, it copies a block of data into the buffer from the file. DMS then locates the specified record in the buffer and copies it into the user record area. DMS uncompresses a compressed record while copying it from the buffer to the user record area. The actual processing of data occurs in the user record area. All of these operations are user-transparent, handled automatically by DMS.

An unmodified Read, such as the one described in the previous paragraph, simply copies data from the data file, and thus does not limit access to that data by another user. If you issue a Read function request with a Hold modifier, a hold is placed on the data item and no other user can access that item of data until the hold is released.

4.5.2 The Write Function Request

The Write function request writes a record from the user record area to the data file. DMS usually performs Write operations using the Segment 2 buffer, but in some special cases DMS writes records directly to the data file. You can code Write function requests to write the initial data to a new file, to write additional records to the end of a relative or consecutive file, or to insert records into a relative or indexed file. In BAM and PAM the Write operation is used to consecutively write 2K byte blocks of data to a data file.

For relative files, you can use the Write operation to either write records sequentially, or write a record according to its relative record number. The type of Write performed is determined by the mode in which you have the file open. A successful Write operation writes a record into an empty slot of a relative file; a Write operation fails if a record is already located in the slot. To overwrite existing records in a relative file, use the Rewrite function request.

For indexed files, the Write operation automatically performs block splits where needed and updates primary index and alternate index key trees. Alternate index key trees are updated as part of the Write operation for all modes except Output mode. In Output mode a file's alternate index key trees are built when the file is closed.

The Write function request is the only permitted record access operation for a file opened in Output mode. Only one user can write records to a file in Output mode. However, if the file is a log file multiple users can open the file in Shared mode for output processing, performing concurrent Write operations on the file.

4.5.3 The Rewrite Function Request

A Rewrite function request writes records that were previously read from the file. A Rewrite follows a Read Hold function request; it rewrites the file record and releases the record hold. You can only use Rewrite in I/O and Shared modes.

In consecutive files, you can only rewrite a record if it is the same length as the original record. For this reason, you cannot rewrite records to a compressed consecutive file.

In relative files, a Rewrite is used to overwrite a record already residing in a record slot. A relative file Rewrite normally follows a Read Hold function request. However, unlike all other file types, you can perform a relative file Rewrite without issuing a previous Read Hold function request.

4.5.4 The Start Function Request

You can use the Start function request in several modes and all access methods. Start function requests perform a variety of pointer positioning routines. You use Start function requests to establish conditions for other function requests rather than to directly access a data record. You can use Start function requests to perform the following operations:

- To position the currency pointer within the file.
- To change the file access mode.
- To hold or release resources.
- To wait for the completion of an I/O operation.

The exact function of each Start operation is defined by its user-supplied modifier. The allowed modifiers differ according to file type, mode, access method, and programming language.

4.5.5 The Delete Function Request

The Delete function request deletes a record from a relative or indexed file. Normally, a record to be deleted must be first read from the file with a Read Hold function request. The deletion of the record releases the record hold. DMS can only perform a Delete on relative or indexed files in I/O or Shared modes.

In indexed files, the Delete operation updates primary index and alternate index key trees as needed to reflect the deletion of the record. DMS closes up space left between the remaining records in a data block following a deletion, and modifies the block length indicator to reflect the block's new length.

In relative files, you normally perform a Delete operation after holding the record with a Read Hold function request. However, unlike indexed files, you can perform a record deletion without performing a Read Hold operation. A relative file deletion resets the slot's record length indicator to zero, making the slot immediately available to a Write operation. A relative file record deletion does not zero-fill the space occupied by the deleted record.

4.6 BUFFERING AND PACKING METHODS

DMS provides two strategies that can improve the efficiency of data file processing: buffering and packing density. These strategies are optional methods of maximizing performance in specific record processing situations.

Buffering strategies allow you to reduce I/O overhead in file processing by establishing temporary data storage buffers. The use of buffers can speed data file processing considerably. DMS provides default buffering in RAM and BAM. In RAM the number of main memory buffer blocks depends on the type of file being processed: one buffer for a consecutive or relative file, two for an indexed file, and three buffer blocks for an alternate indexed file.

You can allocate additional buffer blocks by using one of the two DMS buffering strategies:

- The large buffer strategy for consecutive or relative files in RAM, or for any file type in BAM.

- The buffer pooling strategy for RAM access to indexed files.

You can establish a main memory buffer of up to nine buffer blocks using the large buffer strategy. A large buffer speeds sequential processing of records on a consecutive file or a relative file by reducing the number of I/O operations to storage devices.

For indexed file processing, you can set up a pool of up to sixty buffer blocks in Segment 2. The buffers in this buffer pool retain the upper level index blocks of key trees, speeding record location by primary or alternate key value.

A indexed file's packing density refers to the percentage of space in each file block occupied by data when the file was created. A packing density of less than 100% can improve long-term processing efficiency for indexed files that will have many records added to them after they are created.

When you set a packing density to less than 100%, DMS leaves unwritten a portion of the space in each block during the initial creation of the file. DMS uses this reserved space within the blocks for subsequent addition of records to the file in I/O or Shared modes. Placing additional records within existing blocks minimizes block splitting and the establishment of pointers to distant parts of the data file. This results in rapid writing of records to enlarge the file, and minimizes degradation of file processing performance after many new records have been added to a file.

4.7 FILE ACCESS SUMMARY

In order to access data in a DMS file, you must establish a User File Block (UFB) for that file and write file definition parameter values into the fields of the UFB. In high-level languages the compiler creates UFBs automatically; you simply supply the parameter values. Parameter values can be supplied when the program is compiled, or at runtime, but must be made available to the Open operation.

The Open statement allocates space for the creation of new files and buffer areas for file processing. It defines and limits the types of operations DMS can perform on the file, based on the file access method (RAM, BAM or PAM), the mode (Input, Output, I/O, Extend, Shared), and the file and record structure.

You can access records or blocks in open files by means of function requests. The five function requests (Read, Write, Rewrite, Delete, and Start) perform the actual I/O operations on data files. The way in which these function requests are performed is often determined by a function request modifier.

At the conclusion of file access, you should close all open files. You may open and close the same file repeatedly within the same program in most programming languages.

CHAPTER 5
ACCESSING DMS THROUGH HIGH-LEVEL LANGUAGES

5.1 OVERVIEW

The features of DMS disk access are outlined briefly in this chapter for each of the high-level languages: BASIC, COBOL, Fortran 66, PL/I, and RPG II. These features are described in detail in Chapters 6 through 15, using terms and examples from Assembly language. While Assembly language is the most powerful language for DMS access, it is not always a language with which a programmer is completely comfortable. This chapter provides a bridge from the language-independent aspects of DMS to specific programming reference information. It is intended to orient the reader, rather than to supplant the file I/O information found in the language manuals. No attempt is made to describe all of the data management features available from each high-level language; for specific coding details, consult the individual language manuals.

5.2 DMS FUNCTIONS ACCESSIBLE ONLY IN ASSEMBLY LANGUAGE

Although the most frequently used features of DMS are supported by all VS languages, some operations can only be performed using Assembly language. Among these are:

- The BAM and PAM access methods.
- Using the Start function request to change the mode of an open file.
- Directly modifying the bit mask suffix of an alternate indexed file record.
- Reading the currency pointer field in the UFB.
- Reading certain UFB error status fields.

To carry out these operations, you must call an Assembly language subroutine from a high-level language program, and pass the address of the file's UFB to the subroutine. This type of subroutine call can be performed in COBOL and BASIC; the address of the UFB is not accessible in the other high-level languages. This type of UFB "doctoring" should be done with care, as the effects of modifying a UFB field are not always predictable.

In order to discuss high-level language access to the User File Block you should first understand how Assembly language writes to the UFB. Assembly language programs can supply parameters to a file's UFB for use by the Open operation in either of two ways.

The more common method of supplying these parameters is by means of the UFBGEN macroinstruction, which is coded in the STATIC Section of the Assembly program. When the program is assembled, the UFBGEN parameters are written into a User File Block. Use of the UFBGEN macroinstruction and the available parameters for opening disk files is described in Chapter 6. UFBGEN parameters for non-disk files are described in the chapter corresponding to the device accessed.

A second method of supplying parameters to the UFB is to locate the field within the User File Block directly. A UFB field is located either by unique field name established by use of a suffix, or by using base register addressing. A value written to the UFB at runtime overwrites any previously created UFBGEN value. In this way, both fields initialized by UFBGEN, and UFB fields that are not supplied with a UFBGEN parameter can be modified. This direct access to the fields of the User File Block is only available from Assembly language. Further details on addressing the UFB are provided in Chapter 6.

5.3 HIGH-LEVEL LANGUAGE SUPPORT FOR DMS FEATURES

The degree of support for DMS features varies for the different high-level languages. Table 5-1 provides a comparison of the features available in the different languages.

Table 5-1. High-level Language Support for File Types

	BASIC	COBOL	Fortran 66	PL/I	RPG II
Consecutive	X	X	X	X	X
Shared Consec.					X
Relative		X		X	
Primary Indexed	X	X		X	X
Alt. Indexed	X	X			X

5.3.1 Access Modes

DMS provides five access modes: Input, Output, I/O, Extend, and Shared. The Input mode is used for reading a file, the Output and Extend modes are used for writing a file, and the I/O and Shared modes are used for updating (reading and writing) a file.

The equivalents of these modes in the high-level languages are listed in Table 5-2.

Table 5-2. High-level Language Support for File Access Modes

	BASIC	COBOL	Fortran	PL/I	RPG II
Input	Input	Input	Read	Input	Input
Output	Output	Output	Write	Output	Output
I/O	IO	I-O		Update	Update
Extend	Extend	Extend		Output	+ Add
Shared	Shared	Shared			Shared

For more details on the different types of modes, refer to Chapters 4 and 6. The Shared mode is described in greater detail in Chapter 8.

5.3.2 Function Requests

DMS supplies five function requests: Read, Write, Rewrite, Delete, and Start. Language support for these function requests is outlined in Table 5-3.

Table 5-3. High-level Language Support for Function Requests

	BASIC	COBOL	Fortran	PL/I	RPG II
Read	READ	READ	READ	READ	READ
Write	WRITE	WRITE	WRITE	WRITE	EXCPT
Rewrite	REWRITE	REWRITE		REWRITE	EXCPT
Delete	DELETE	DELETE		DELETE	DELET
Start	SKIP	START			SETLL/SETGT

An overview of the use of function requests in each high-level language is provided in this chapter. For additional details on function requests, refer to Chapters 4 and 7.

5.4 SUPPLYING FILE DEFINITION PARAMETERS IN BASIC

To access a DMS file, the program must supply file definition parameters to the User File Block (UFB), where they are made available to the OPEN instruction. In BASIC these parameters are coded in two places: the SELECT statement and the OPEN statement. The BASIC program must contain a SELECT statement for each file opened; the SELECT statement must occur in the program prior to the first OPEN statement for that file. You can open a file several times during a program run, but you can select it only once.

5.4.1 BASIC File Definition Parameters

A DMS file has three names in VS BASIC: an internal identifier used within the program to name the file; an external parameter reference name used for linking and error processing; and a permanent file name which is recorded in the Volume Table of Contents (VTOC) and is the same for all programs accessing the file. The first two file names are coded in the SELECT statement in VS BASIC; you supply the third when the file is opened.

In VS BASIC the internal file identifier is a file number for 1 to 64 preceded by a number symbol (#). You give each file used by the program a unique number. Files can be numbered and accessed in any order. The name in quotes following the file number is the parameter reference name (or PRNAME). See Example 5-1.

The device type and the file organization are both specified with a single term in the SELECT statement. The four choices are:

INDEXED	An indexed or alternate indexed disk file, since only disk files can be indexed.
CONSEC	A consecutive disk file. Consecutive files for other devices are specified by the device type.
TAPE	A consecutive magnetic tape file.
PRINTER	A consecutive print file.

You specify the file's permanent file, library, and volume names in the OPEN statement. DMS matches the SELECT and OPEN statement by means of the file number. You specify these names only the first time the file is opened.

Example 5-1. BASIC File Defining Parameters

```
SELECT #4, ''ZOOPRNAME'', CONSEC.
.
.
.
OPEN #4, FILE=ZOOFILE, LIBRARY=ZOOLIB, VOLUME=ZOOVOL.
```

5.4.2 BASIC File Allocation Parameters

In order to allocate space for a data file, you must provide DMS with the size of the records in the file and the estimated number of records to be initially written to the file. In VS BASIC, place the record size (or RECSIZE) in the SELECT statement, and the number of records (SPACE) in the OPEN statement, as follows:

```
SELECT #4, RECSIZE=80.
    .
    .
    .
OPEN #4, SPACE=100.
```

If a file contains variable length records, the RECSIZE indicates the maximum record size. Variable length record files precede the RECSIZE clause with VAR; compressed record files precede RECSIZE with VARC, as follows:

```
SELECT #4, VARC, RECSIZE=80.
```

5.4.3 BASIC Primary and Alternate Index Key Parameters

VS BASIC provides full support for indexed and alternate indexed files. You provide all the necessary information for creating these indexed disk files in the SELECT statement. An indexed file with a primary key is logically defined as follows:

```
SELECT #7, INDEXED, KEYPOS=1, KEYLEN=10.
```

This SELECT statement establishes a primary key 10 bytes in length beginning at the first byte of each record. KEYPOS and KEYLEN are equivalent to the UFBGEN parameters KPOS and KSIZE used in Assembly language.

To access alternate indexed keys, you must supply four items of information: which of the sixteen possible alternate key paths is being defined, the starting position of the alternate key, the length of the alternate key, and whether the key path should allow duplicate alternate key values. The SELECT statement in Example 5-2 shows the coding for a primary key and three alternate keys:

Example 5-2. Selecting an Alternate Indexed File in BASIC

```
SELECT #7, INDEXED, KEYPOS=1, KEYLEN=10,
    ALT KEY 1, KEYPOS=11, KEYLEN=5,
    ALT KEY 2, KEYPOS=25, KEYLEN=1, DUPS,
    ALT KEY 9, KEYPOS=40, KEYLEN=10, DUPS.
```

In order to place individual records on alternate index key paths, the BASIC programmer uses a MASK function statement prior to writing a record to the file.

5.4.4 BASIC File Efficiency Parameters

DMS uses two methods for improving the efficiency of file processing: buffering, and packing density. File efficiency parameters take a default value. To set values other than the defaults, the BASIC programmer codes parameters in the OPEN statement. These parameters only need to be specified the first time the program opens the file.

I/O efficiency of a consecutive file can often be improved by specifying a buffer of larger than one block. A Segment 2 buffer of up to 9 blocks can be created by specifying:

```
OPEN #7, BLOCKS=n
```

where n can take a numeric value from 1 to 9.

The BLOCKS parameter is only used for consecutive files. For indexed files, you can establish a buffer pool by means of the SELECT POOL statement. This is not the same statement as the SELECT statement used to define the file. Code both the SELECT file and SELECT POOL statements to create an indexed file with a buffer pool. The two SELECT statements are shown in Example 5-3.

Example 5-3. BASIC Coding for Buffer Pooling

```
SELECT #4, ''ZOOFILE'', INDEXED,
...
SELECT POOL#4, BLOCKS=16
```

where #4 is the file number of the file(s) using the buffer pool, and BLOCKS takes a value from 3 to 60 buffer pool blocks.

Packing densities of less than 100% should be established for an indexed file if the file is expected to increase significantly in size. Index and data block packing densities are set in the OPEN statement in BASIC, as follows:

```
OPEN #4, DPACK=80, IPACK=95
```

where DPACK corresponds to the data block packing density, and IPACK to the index block packing densities. Densities are expressed in percentage of the block available for record writing during file creation.

For further details on buffering and packing density, refer to Chapter 9. More comprehensive information on BASIC is found in the VS BASIC Language Reference.

5.4.5 The BASIC CALL Statement and the UFB

The User File Block is not directly accessible in BASIC, limiting the user's abilities to set file parameters. These limits can be circumvented by using the CALL statement to call a subroutine written in Assembly language and pass to it the address of the file's UFB. The Assembly language subroutine sets the desired UFB field dynamically, then issues a RETURN to the BASIC program. The CALL statement appears as follows:

```
CALL ''zoosub'' ADDR(#4)
```

where zoosub is the name of the Assembly subroutine and ADDR(#4) locates the address of the UFB for BASIC file #4.

5.5 DMS RECORD ACCESS FROM BASIC

You must supply one of the five following file access modes in BASIC as part of the OPEN statement: Input, Output, I/O, Extend, and Shared. You must specify a file access mode every time you open a file; to change the mode you must close the file and then reopen it.

BASIC uses five statements, or function requests, to locate and access records: READ, WRITE, REWRITE, DELETE, and SKIP. The first four are similar in function to their Assembly language counterparts described in greater depth in Chapter 7. SKIP is analogous to the START BEGIN and START SKIP Assembly language function requests.

Use the SKIP statement to locate records on a consecutive file by position relative to the current location pointer. DMS uses the RECORD clause of the READ statement to locate a consecutive file record by relative record number.

The KEY clause of the READ statement provides flexible indexed file record access. Both primary and alternate keys can be located by exact or approximate values. You can locate records using two types of approximate key values: you can supply only the first character(s) of the key value to retrieve the first record with that partial key, or specify a key value and request the first key value larger than the one you specified. By setting the KEY field to zeros, DMS searches for the first record in the file.

A WRITE statement is used to place a record in a file. A WRITE can be used in Output mode to create a new file, or in Extend mode to add records to the end of a consecutive file.

If you want to update a record already in a file, open the file in I/O or Shared mode, issue a READ HOLD to read the record, update the fields of the record, and return the updated record to the file using a REWRITE statement.

The DELETE statement is used to delete a record from an indexed file. The DELETE statement must be preceded by a READ HOLD.

5.6 SUPPLYING FILE DEFINITION PARAMETERS IN COBOL

VS COBOL provides a very powerful set of data management functions. The programmer using VS COBOL has access to some I/O features (such as advanced sharing) unavailable in most other high-level languages, and that require considerable knowledge to perform in Assembly language.

COBOL supports relative files; it is the only VS high-level language that currently supports relative files. COBOL also supports the DMS/TX transaction recovery and multiple record holding functions, as described in the VS DMS/TX Reference.

Disk file processing parameters can be divided into four functional groups: file definition parameters specifying file names and types, file allocation parameters used to compute the space requirements of a file, parameters used to establish index and alternate index keys, and optional parameters for file efficiency.

5.6.1 COBOL File Definition Parameters

The parameters used to define a file are coded in the Environment Division of a COBOL program in the File-Control Section. Within the File-Control section is a file-control entry (SELECT) for each file accessed by the program. A file-control entry specifies the internal name of the file, the logical file name by which the file is referred within the program.

Once you have selected a file, you can specify its parameters in any order. The first line of the file-control entry shown in Example 5-4 is an ASSIGN clause containing the parameter reference name (or PRNAME) and the type of I/O device for the file. The parameter reference name is the program's external name for the file, and is used in error processing. The PRNAME is followed by a device type keyword. Device type options are "DISK", "DISPLAY" (for workstation files), "PRINTER" and "TAPE". The PRNAME and the device type keyword are always in quotes.

The next phrase in the example, an ORGANIZATION clause, stipulates the structure of the file. Specify ORGANIZATION IS SEQUENTIAL for consecutive files on disk and files for any non-disk device. The keyword "SEQUENTIAL" in COBOL refers to consecutive files. Specify ORGANIZATION IS RELATIVE for relative files. Specify ORGANIZATION IS INDEXED for indexed and alternate indexed files on disk.

You can specify the ACCESS MODE as being either SEQUENTIAL, RANDOM, or DYNAMIC. You can also specify the key field name by including a RELATIVE KEY clause for consecutive or relative files, or a RECORD KEY clause for indexed files.

Example 5-4. COBOL Environment Division Coding

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT FIRST-ZOO-FILE
    ASSIGN TO ''ZOO1'', ''DISK'',
    ORGANIZATION IS SEQUENTIAL
    ACCESS MODE IS DYNAMIC
    RELATIVE KEY IS ZOO-KEY-FIELD.
SELECT SECOND-ZOO-FILE
```

Not all file identification parameters are specified in the Environment Division. The permanent file, library, and volume names of the file are recorded in the file description entry in the File Section of the Data Division, and are referenced by the internal name specified in the file-control entry:

Example 5-5. COBOL Assignment of Permanent File Names

```
DATA DIVISION.
FILE SECTION.
FD  FIRST-FILE
    VALUE OF FILENAME IS ''ZOOFILE''
             LIBRARY IS ''ZOOLIB''
             VOLUME IS ''ZOODISK''
             .
             .
             .
```

5.6.2 COBOL File Allocation Parameters

In order to create a new file, DMS must allocate sufficient space for that file. This requires supplying two numeric parameters to the UFB -- the record size (RECSIZE) and the anticipated number of records to be initially placed in the file (NRECS). For relative files, the NRECS field is the number of record slots to be placed in the file's first allocation. In COBOL these parameters are coded in the DATA DIVISION file descriptor (FD) as shown in Example 5-6.

Example 5-6. COBOL Space Allocation Parameters

```
DATA DIVISION.
FILE SECTION.
FD  ZOO-FILE
    RECORD CONTAINS 80 CHARACTERS
    VALUE OF SPACE IS ZOO-N-REC
    .
    .
    .
WORKING-STORAGE.
01  ZOO-N-REC    BINARY VALUE 100.
```

Example 5-6 allocates the space for 100 fixed length records, each record 80 bytes in length. The record size (RECORD CONTAINS) clause has two variants:

RECORD CONTAINS n TO m CHARACTERS
[used for variable length records]

RECORD CONTAINS nn COMPRESSED CHARACTERS
[for compressed variable length records]

Remember that relative files cannot contain compressed characters.

5.6.3 COBOL Primary and Alternate Index Key Parameters

If the file-control entry specifies ORGANIZATION IS INDEXED, you must specify the names of the primary key field and any alternate index key fields in the RECORD KEY and ALTERNATE RECORD KEY clauses. THE RECORD KEY field is a mandatory field that supplies the internal name of the primary index key field. The name of this key field reappears in the Data Division file description entry (FD), where the size and location of the primary key are listed along with the picture clauses for the other fields in the record.

Example 5-7. COBOL Definition of Primary Keys

```
ENVIRONMENT DIVISION.
    SELECT ZOO-FILE
        ORGANIZATION IS INDEXED
        RECORD KEY IS ANIMAL-NAME
        ...
DATA DIVISION.
FILE SECTION.
FD  ZOO-FILE
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS ZOO-RECORD.
01  ZOO-RECORD.
    05  ENCLOSURE     PIC X(10).
    05  ANIMAL-NAME   PIC X(12).
    05  FILLER        PIC X(58).
```

COBOL provides full support for alternate indexed files. In order to create or access alternate indexed files the system must create an AXD1 block in Block 0 of the data file, and set the key path bits on the two-byte suffix mask at the end of each data record. You invoke these operations by defining the key paths in the ENVIRONMENT DIVISION, and stipulating the key size and location in the DATA DIVISION file descriptor.

Establishing a primary record key and three alternate record keys is shown in Example 5-8. Note that the second and third alternate index key paths allow duplicate key values.

Example 5-8. COBOL Coding for a File with Alternate Keys

```
ENVIRONMENT DIVISION.
FILE-CONTROL.
    SELECT ZOO-ALT-FILE.
    ASSIGN TO ''ZOO-PRNAME''  ''DISK''
    ...
    RECORD KEY IS ANIMAL-NAME
    ALTERNATE RECORD KEY
        01  IS SCIENTIFIC-NAME
        02  IS SEX              WITH DUPLICATES
        03  CARNIVORE-FLAG      WITH DUPLICATES.
        .
        .
        .
DATA DIVISION.
FILE SECTION.
FD  ZOO-ALT-FILE
01  ZOO-REC.
    05  ANIMAL-NAME      PIC X(10).
    05  PET-NAME         PIC X(10).
    05  SCIENTIFIC-NAME  PIC X(20).
    05  CARNIVORE-FLAG   PIC X.
    05  SEX              PIC X.
```

A COBOL programmer cannot set the alternate index record mask bits directly. To place a record on a subset of the available keys you must establish that parallel set of alternate index keys in the file-control entry, and provide an alternate record description as shown in Example 5-9. Records written with the first file descriptor are placed on three alternate key paths. Records written with the second file descriptor are placed on two alternate key paths.

Example 5-9. COBOL Coding for Records on Different Alternate Keys

```
ENVIRONMENT DIVISION.
...
    ALTERNATE RECORD KEY
    01  IS SCIENTIFIC-NAME
    01  IS SCIENTIFIC-NAME-1
    02  IS SEX           WITH DUPLICATES
    02  IS SEX-1         WITH DUPLICATES
    03  CARNIVORE-FLAG   WITH DUPLICATES

DATA DIVISION.
FILE SECTION.
FD  ZOO-ALT-FILE
01  ZOO-REC-3-ALT.
    05  ANIMAL-NAME        PIC X(10).
    05  PET-NAME           PIC X(10).
    05  SCIENTIFIC-NAME    PIC X(20).
    05  CARNIVORE-FLAG     PIC X.
    05  SEX                PIC X.
01  ZOO-REC-2-ALT.
    05  ANIMAL-NAME-1      PIC X(10).
    05  PET-NAME-1         PIC X(10).
    05  SCIENTIFIC-NAME-1  PIC X(20).
    05  CARNIVORE-FLAG-1   PIC X.
    05  SEX-1              PIC X.
```

5.6.4 COBOL File Efficiency Parameters

COBOL supports the full range of performance enhancement methods. You can establish large buffers for consecutive file or relative file processing in the file-control entry by using a BUFFER SIZE IS n clause with a multiple of 2048 bytes. You can establish a buffer pool for indexed files by specifying a RESERVE n AREAS clause in the file-control entry, in conjunction with the SAME AREA clause. In these clauses, n is a number of blocks between 3 and 60. Several files can share a buffer pool.

Packing densities are used with indexed files to leave blank space in the file blocks for subsequent updates. You establish these density percentages in the DATA DIVISION file descriptor by specifying:

```
FD  ZOO-REC.
    VALUE OF DATA AREA IS n
             INDEX AREA IS m
```

where n and m correspond to packing density percentages with a default of 100.

5.6.5 The COBOL CALL Statement and the UFB

The UFB is not directly accessible in COBOL, limiting your control over file parameters. You can circumvent these limits by using the CALL statement to call a subroutine written in Assembly language and pass to it the address of the file's UFB. The Assembly language subroutine sets the desired UFB field dynamically, then issues a RETURN to the COBOL program. The CALL statement appears as follows:

```
CALL ''ZOOSUB'' USING ZOO-FILE
```

where zoosub is the name of the Assembly program and zoo-file is the COBOL internal file name used in the SELECT clause.

For further details on establishing file parameters in COBOL, see Chapter 9 of the VS COBOL Reference Manual.

5.7 DMS RECORD ACCESS FROM COBOL

Record access statements in VS COBOL bear many similarities to access statements in Assembly language. COBOL specifies the mode as part of the OPEN statement, and uses the five standard Open modes: Input, Output, I-O (note hyphen), Extend, and Shared. You must specify an Open mode every time you open a file; to change the mode you must close the file and then re-open it.

COBOL uses five PROCEDURE DIVISION statements to locate and access records: READ, WRITE, REWRITE, START, and DELETE. These five statements are referred to as function requests in Chapter 7 where their use is described in depth.

COBOL does not use a READ REL or READ KEYED statement. Instead, it uses the ACCESS MODE clause of the SELECT statement in combination with the READ function request to locate records within a file. This ENVIRONMENT DIVISION statement determines the method for locating records in a file for the entire program. The ACCESS MODE can be SEQUENTIAL, RANDOM, or DYNAMIC. SEQUENTIAL access limits record access to reading record-by-record from the beginning of the file. This is the only access method permitted for magnetic tape. If you specify RANDOM for a sequential or relative file, the READ statement accesses the record specified by a relative key value (relative record number). If you specify RANDOM for an indexed file, the READ statement accesses the record specified by the primary or alternate record key value. DYNAMIC access mode allows records in a file to be read either sequentially or randomly.

The various access methods and types of READ statements are shown in Table 5-4.

Table 5-4. READ Statements in COBOL

Environment Division File-Control Entry	Locate and Read Sequentially	Locate and Read by Relative Record Number
ORGANIZATION IS SEQUENTIAL ACCESS MODE IS SEQUENTIAL	READ	
ORGANIZATION IS SEQUENTIAL ACCESS MODE IS RANDOM		READ
ORGANIZATION IS SEQUENTIAL ACCESS MODE IS DYNAMIC	READ NEXT	READ

Environment Division Select Statement	Locate and Read Sequentially	Locate and Read by Relative Record Number
ORGANIZATION IS RELATIVE ACCESS MODE IS SEQUENTIAL	READ	
ORGANIZATION IS RELATIVE ACCESS MODE IS RANDOM		READ
ORGANIZATION IS RELATIVE ACCESS MODE IS DYNAMIC	READ NEXT	READ

Environment Division Select Statement	Locate and Read Sequentially	Locate and Read by Primary Key Value
ORGANIZATION IS INDEXED ACCESS MODE IS SEQUENTIAL	READ	
ORGANIZATION IS INDEXED ACCESS MODE IS RANDOM		READ
ORGANIZATION IS INDEXED ACCESS MODE IS DYNAMIC	READ NEXT	READ

To access a record by alternate key value, you issue a READ statement with a KEY IS clause, establishing the key of reference. The key of reference is the primary or alternate index key path on which DMS performs the read. The default key of reference is the primary key. Once you establish a key of reference, DMS performs all READ function requests along that path until: [1] you issue a KEY IS clause to establish another key of reference [2] the file is closed, causing the key of reference to default to the primary key, or [3] you issue a START function request, resetting the key of reference to another key.

In COBOL the START statement is used to locate records in indexed files by primary or alternate key value. You can specify an approximate key value; you can use the START function request to select the first record with a higher key value, or to match a partial key (consisting of the first character(s) of the key) against the file records' keys. A similar use of the START function request permits you to access relative file records by Relative Record Number.

In COBOL the HOLD statement is used to hold an entire file or a group of records within a file that have a common generic key. You can simultaneously hold multiple resources by using a HOLD LIST statement. Held resources cannot be accessed by other users in Shared mode.

A WRITE statement is used to place a record in a file. You can use a WRITE in Output, I-O and Shared mode for indexed files, and in Output, Shared or Extend mode for relative or consecutive files.

If you want to update an existing record in a consecutive or indexed disk file, open the file in I-O or Shared mode, issue a READ WITH HOLD to read the record, update the data and return the updated record to the file using a REWRITE statement. In a relative file opened in sequential access mode, you first read a record, then issue a REWRITE to overwrite the record. In a relative file opened in the random or dynamic access mode, you specify the relative record number of the record to be overwritten in the RELATIVE KEY field, then issue a REWRITE to overwrite the record.

The DELETE statement is used to delete a record from a relative or indexed file. In a relative file opened in sequential access mode, you first locate a record using a READ or START function request, then issue a DELETE to scratch the record. In a relative file opened in the random or dynamic access mode, you specify the relative record number of the record to be deleted in the RELATIVE KEY field, then issue a DELETE to scratch the record. In an indexed file opened in sequential or dynamic access mode, the DELETE statement must be preceded by a READ WITH HOLD.

5.8 SUPPLYING FILE DEFINITION PARAMETERS IN FORTRAN

File processing in Fortran 66 differs considerably from the other languages supported by the VS in that Fortran does not use an OPEN statement. Files are neither opened nor closed, nor are their parameters set forth in a SELECT statement. These differences are more apparent than real. When the first READ or WRITE statement for a file occurs in a program, the program performs the functional equivalent of an OPEN statement. DMS displays a series of screens that request many of the same parameters supplied to an OPEN statement in other programming languages. DMS records the parameter values you input to these screens in the UFB, where they are available the next time the program accesses the file.

5.8.1 Fortran File Definition Parameters

You define a file by supplying a logical unit number to the file. This logical unit number is the file's internal name, used whenever your program refers to that file. Each READ, WRITE, BACKSPACE or REWIND statement must contain a logical unit number. This number can range from 0 to 64, although certain unit numbers (e.g., 0, 5, 6) are reserved for specific device types.

The first appearance of a particular logical unit number causes the system to display a series of screens requesting additional information about the file. These screens are described in Appendix D of the VS FORTRAN Reference manual. A request for the permanent file name, library and volume is displayed for each file defined by a logical unit number. The type of storage device (DISK, WS, or TAPE) is also requested.

5.8.2 Fortran File Allocation Parameters

The data entry screen for the permanent file, library, and volume names also requests an estimated number of records to be written to the file. A second screen requests the number of bytes in each record (RECSIZE). DMS uses these two numbers to calculate the primary extent allocation for the file.

Fortran processing supports variable length records. Again, the parameters are presented to you via a screen display. You supply VAR=YES, and a MAXSIZE equal to the length of the largest record. The primary extent allocation is calculated based on the MAXSIZE. VS Fortran does not support record compression.

5.8.3 Limitations on Fortran Processing of DMS Files

Fortran is primarily a language for scientific calculations, rather than for the storage and retrieval of data records. As such, there are several DMS file support features that are not available directly from Fortran, and are accessible only by calling a subroutine in Assembly or some other VS language.

Fortran 66 does not support indexed or alternate indexed files. In order to read records from an indexed file, you must first copy the records into consecutive file format or use a subroutine in another high-level language.

Other common DMS features, such as packing densities and buffer pooling, are used only with indexed files, and are therefore not supported by Fortran 66.

5.9 DMS RECORD ACCESS FROM FORTRAN

Fortran 66 does not use OPEN or CLOSE statements. The first READ or WRITE operation performed on a file initiates a file opening routine.

Only consecutive files are accessible using Fortran 66. Two file positioning statements are available: BACKSPACE and REWIND. The BACKSPACE statement moves the file pointer to the previous record. The REWIND statement closes the file and repositions the file pointer to the first record in the file.

All file access is performed using the READ and WRITE statements. The READ statement reads the next record from the consecutive file, or the record pointed to by the BACKSPACE or REWIND statements. The WRITE statement writes new records to a file; if the file is pre-existing, records are written to the end of the file. Fortran 66 cannot update records.

5.10 SUPPLYING FILE DEFINITION PARAMETERS IN PL/I

The PL/I programmer has the choice of coding most file parameters in either the DECLARE statement or the OPEN statement. File parameters placed in the DECLARE statement remain unchanged throughout the program. Parameters not declared, but coded only in the OPEN statement govern file access until that file is closed. You can then reopen a file with different parameter values. If you code a parameter in both a DECLARE and an OPEN statement, the two values must match. A conflict between DECLARE and OPEN parameter values results in a runtime error.

Code an ENVIRONMENT attribute for each file accessed by the PL/I program. You can code this attribute as part of either the file's DECLARE statement or as part of each OPEN statement, or in both places. The keyword ENVIRONMENT is followed by parentheses, containing a series of parameters (items) separated by commas. Each item is in the form:

```
Keyword = value
```

which is the same form as the UFBGEN parameters in Assembly language. The value portion of the item should be in quotes unless the value is stipulated as an integer.

5.10.1 PL/I File Definition Parameters

Since the DECLARE and OPEN statements are used for many other things than DMS file access, it is necessary to specify the words FILE and RECORD to indicate that the named entity is a DMS file, accessed by data records. You can do this using either of the following statements:

```
DECLARE filename FILE RECORD ENVIRONMENT...
```
or
```
OPEN FILE(filename) RECORD ENVIRONMENT...
```

The definition of the file is completed by coding an ENVIRONMENT attribute for either the DECLARE or OPEN statement. The first item in the ENVIRONMENT attribute, shown in Example 5-10, is the TITLE name, also known as the parameter reference name or PRNAME. This is an external file name used for calling subroutines and error processing. The next item, DEVICE=DISK, makes clear that this is a disk file, rather than a TAPE, PRINTER, or WS (workstation) file.

The system records the name of a disk file in the disk's VTOC as a permanent name used by all programs that access that file. In PL/I the permanent file, library, and volume names are items in the ENVIRONMENT attribute, as shown in Example 5-10.

An ORGANIZATION item allows you to choose either CONSECUTIVE or INDEXED file structure. CONSECUTIVE is the default, and the only file structure permitted for non-disk files. The FILETYPE item identifies special use files, such as PRINT, PROGRAM or workstation (WS) files. In Example 5-10, this item is supplied with the default value ANY.

5.10.2 PL/I File Allocation Parameters

In order to allocate space for a data file, it is necessary to provide DMS with the size of the records in the file, whether the records are fixed or variable length, and the estimated number of records to be initially written to the file. In VS PL/I these parameters are supplied as items in the ENVIRONMENT attribute. The record length (RECLEN) and the number of records (NRECS) are numeric values. If the record length varies, the RECLEN designates the maximum record size. The items showing that the records are variable length or compressed are in Boolean notation, where a value of '1'B indicates a Yes or True statement. If a file is compressed, it is not necessary to indicate that its records are variable length.

Example 5-10. PL/I ENVIRONMENT Attribute

```
ENVIRONMENT(TITLE='ZOOPRNAME', DEVICE='DISK', FILENAME='ZOOFILE',
            LIBRARY='ZOOLIB', VOLUME='ZOODISK',
            ORGANIZATION='INDEXED', FILETYPE='ANY',
            RECLEN=80, NRECS=100, COMPRESSED='1'B)
```

5.10.3 PL/I Primary and Alternate Index Key Parameters

If a data file resides on a disk and ORGANIZATION='INDEXED', the file must also have two other ENVIRONMENT items: the location of the primary key (KEYPOS) and the length of the primary key (KEYLEN). Specify KEYPOS as a byte position counting from zero, KEYLEN as the byte length of the longest primary key.

Alternate indexed files are not fully supported by VS PL/I. It is possible, however, for a PL/I program to read an alternate indexed file sequentially or by primary index key. Updating alternate indexed files is not recommended, because updates to the alternate index trees are not performed from PL/I.

5.10.4 PL/I File Efficiency Parameters

The user program can often improve the processing of consecutive files by increasing the size of the Segment 2 buffer. The BUFSIZE item in the ENVIRONMENT attribute specifies the size of this buffer. The default value of this item is 2048, which is the number of bytes in one block. BUFSIZEs larger than 2048 must be multiples of 2048 up to 18,432, the number of bytes in nine blocks.

VS file efficiency methods used for indexed and alternate indexed files are not supported from VS PL/I.

5.11 DMS RECORD ACCESS FROM PL/I

PL/I uses an OPEN statement to establish file access attributes for record processing. If you do not code an OPEN statement, the first READ, WRITE, or REWRITE statement automatically opens the file with default access attributes. The OPEN statement attributes include the file access mode and the file access method. You can specify the ENVIRONMENT attribute in either the DECLARE or OPEN statements.

The ENVIRONMENT statement includes a DISPOSITION item that can take the values OLD, NEW, or ANY. A pre-existing file is an OLD file; a file created during the program run is a NEW file. ANY allows a file to be both created and modified during the same program run. DISPOSITION=OLD, when combined with the Output file access mode, extends a consecutive file by adding records.

The three file access modes, INPUT, OUTPUT, and UPDATE govern the type of record access performed while the file is open. The UPDATE mode corresponds to the DMS I/O mode; if UPDATE mode is specified with an ENVIRONMENT attribute that contains a SHARED='1'B item, file access is in the DMS Shared mode.

The three file access methods -- SEQUENTIAL, DIRECT, and KEYED -- establish the location method for accessing records. The possible combinations are as follows:

SEQUENTIAL	Sequential access to records in consecutive or indexed files
SEQUENTIAL KEYED	Access to records in consecutive files by relative record numbers
DIRECT KEYED	Access to records in indexed files by primary key value

The relative record number or primary key value sought is coded in the KEY option of the READ statement. If you do not specify a KEY, DMS performs a sequential READ.

PL/I provides WRITE, DELETE, and REWRITE statements. The WRITE statement writes a record to a file. The DELETE statement deletes a record from an indexed file; you can use the KEY option of the DELETE statement to locate the record to be deleted. To rewrite a record in an UPDATE file, read the record from the file, update the record data, then use the REWRITE statement to update the record in the file. For a PL/I REWRITE operation, the READ statement does not require a HOLD clause.

There is no START statement or its equivalent in PL/I. Records must be located by primary key value or by relative record number. Files cannot be accessed by alternate key values.

The PL/I programmer can also perform file update using the PUT and GET statements. These statements can only be used for sequential processing of records.

5.12 SUPPLYING FILE DEFINITION PARAMETERS IN RPG II

An RPG II program is divided into functional sections known as specifications. Each of these specifications is written using its own coding form, with numbered columns assigned to particular items. RPG II programming requires you to place the appropriate letter code, number, or word into the correct column.

RPG II is unique among the high-level languages in that it permits you to share consecutive disk files. File sharing is described in detail in Chapter 8. RPG II also provides support for the file sharing and recovery features of DMS/TX. DMS/TX is described in the VS DMS/TX Reference.

You supply file definition parameters to the file's UFB by coding them in the File Description Specification (or F Spec), the first section of an RPG II program. Each file is described in a single line, using the column positions to stipulate the file parameters. Since RPG II is a powerful file-handling language, there are more file and record parameters required for a file than are used by DMS. This chapter passes over these RPG II-specific parameters and focuses on the file definition parameters supplied to DMS for disk file processing.

5.12.1 RPG II File Definition Parameters

An internal file name identifies all references to a data file within a RPG II program. Code this file name in Columns 7 to 14 of the File Description Specification. The parameter reference name (PRNAME) defaults to the internal file name.

The permanent file, library, and volume names recorded in the volume table of contents (VTOC) are not compiled in an RPG II program itself; instead, you specify these parameters at execution time by means of a GETPARM screen or procedure language statements.

You denote the type of device used for file I/O in columns 40 to 46 of the File Description Specification. If the file device is a disk (DISK in columns 40 - 43), the file can be structured as an indexed, alternate indexed, or consecutive file; if the file is on a device other than a disk, only consecutive structure is possible. The file structure is recorded in column 32 of the F Spec. Consecutive files use column 32 for other purposes as well, as described later in this section.

5.12.2 RPG II File Allocation Parameters

In order to allocate space for a data file, you must specify the size of the records in the file, whether that record size is fixed or variable, and the estimated number of records to be initially written to the file. In RPG II the record size and its variability are included in the F Spec. Column 24 accepts the record size (RECSIZE), column 19 indicates whether the record size is fixed (F) or variable (V), and column 74 specifies compression. If you specify compression for a file, you must also specify variable length records and provide the maximum uncompressed record size in the RECSIZE field.

5.12.3 RPG II Primary and Alternate Index Key Parameters

You must stipulate the primary index key length and position for all indexed or alternate indexed files. Columns 29 and 30 of the File Description Specification are reserved for the primary key length, Columns 35 through 38 for the starting position of the primary key in each record.

RPG II provides full support for alternate indexed files by means of a separate specification, the Alternate Index Specification, or A Spec. Alternate Index Specifications appear in the program immediately after the File Description Specifications. Each file that stipulates an alternate indexed file structure (an "A") in Column 32 of the File Description Specification requires an Alternate Index Specification.

The alternate indexed file is identified by the internal filename in Columns 7 - 14. Four parameters define an alternate key path: the key path number (values from 1 to 16), the alternate key length, the alternate key starting position, and a flag allowing or disallowing duplicate alternate key values.

Each Alternate Index Specification line has room for the parameters of four alternate key paths. Space for defining more alternate key paths is reserved by repeating the internal file name in Columns 7 - 14 of additional Alternate Index Specification lines. Thus you can establish the maximum of 16 alternate key paths in four successive lines. You need specify only those alternate key paths accessed by the program.

In order to place new records on an alternate key path you must list the alternate path numbers in Columns 45 to 70 of the Output Specification sheet.

5.12.4 RPG II File Efficiency Parameters

The user program can often improve the processing of consecutive files by increasing the size of the Segment 2 buffer. In RPG II this is carried out by specifying a number in File Description Specification Column 32, the file structure code column. Since the default file structure is consecutive, this field does double duty for consecutive files as the Segment 2 buffer size column. The size can range from one (the default) to nine buffer blocks, each 2048 bytes long.

VS file efficiency methods used for indexed and alternate indexed files are not supported from VS RPG II.

5.13 DMS RECORD ACCESS FROM RPG II

The RPG II programming language is unique in that it does not normally use OPEN, CLOSE, READ, or WRITE statements for file access. In RPG II, access to the records in a file is performed as part of the program cycle. The first step of the cycle is to open an input file and read a record. Records are read in either sequential order or in the sequence specified by ADDROUT or KEYOUT (see below). If you specified a file as an output file, the record is written to that file. RPG II then begins the cycle over, reading the next input record. At end-of-file, the system closes the input and output files.

The file mode specified in Column 15 of the File Description Specification controls this cyclic reading, updating, and writing of records. Since each DMS file has only one File Description Specification line, and a file cannot be opened and closed by the programmer, an RPG II program can only access a file in a single mode.

The three available disk file modes are Input, Output, and Update (I, O, and U). Any of these modes can be used for extending a file by specifying an ADD. An Update mode indexed file can be shared by placing an S in Column 73 of the File Description Specification. In RPG II, multiple users can also share consecutive disk files in Input or Update modes. You specify a shared consecutive file by placing an S in Column 73 of the File Description Specification. HOLD and HOLDL statements allow multiple users to hold single or multiple records when updating shared files. You can also specify consecutive disk files in Output mode as shared log files.

Column 28 of the File Description Specification determines whether DMS is to access records sequentially, or by a relative record number or indexed record key value. You can use the CHAIN operation to locate records by relative record number or key value.

In addition to the automatic read and write processing, there are operations that force operations outside of their normal cycle. The READ and CHAIN operations force the reading of a record; the EXCPT operation forces the execution of the output portion of the cycle, which performs write and rewrite operations.

The DELET [note spelling] operation deletes the last record read from an index or alternate indexed file.

RPG II has two equivalents to the DMS START function request for indexed files. The SETLL and SETGT operations position the location pointer to a record by primary key value. Either instruction can be followed by an explicit READ or a cycle-generated read, initiating sequential processing of the file from that point.

The SETLL operation sets the lower limit for sequential processing of indexed records. The file location pointer is positioned by SETLL to the specified primary key value, and sequential processing begins at that point. The SETGT operation positions the location pointer to the first record with a primary key greater than the SETGT-specified value. The user can locate the first file record by setting the SETGT value to zeros.

5.13.1 RPG II Special Record Processing Options

ADDROUT and KEYOUT allow you to access a data file by any field within the records. In DMS this function would be performed by establishing an alternate index path for the field, then performing READ KEYED access along that key path. RPG II provides an option to the use of alternate indices for this purpose.

RPG II uses ADDROUT (address output) and KEYOUT (key output) files to process all the records in a data file in a sequence other than consecutive. It does not use them to locate individual records. It can process records in ascending ASCII sequence by any field within the records. An ADDROUT or KEYOUT sequence file governs the sequence of record processing.

Use the ADDROUT file type for processing consecutive files with fixed length records. Run the SORT utility with the ADDROUT option, and specify one of the fields of the records as a sort field. The output of the SORT utility is an ADDROUT file consisting of a list of the consecutive file's relative record numbers in the sequence dictated by the sort. You then input this ADDROUT file to your RPG II program to prescribe the sequence for reading the records in the original data file. The system performs read operations randomly by the relative record numbers in the ADDROUT file.

Use the KEYOUT file type with indexed files. It performs a sort by a user-specified field and builds a KEYOUT file. Run the SORT utility with the KEYOUT option, and specify a field other than the primary key as a sort field. The output of the SORT utility is a KEYOUT file of the primary key values of each record in the file. The KEYOUT file is a consecutive file; its records are in the sequence established by the SORT utility. You then input this KEYOUT file to your RPG II program. When the system processes the original data file, it reads the records in the order dictated by the KEYOUT file. Records are accessed randomly by primary key value, using the table of primary keys in the KEYOUT file to determine the reading sequence.

CHAPTER 6
DEFINING DMS DISK FILES / THE USER FILE BLOCK

6.1 OVERVIEW -- THE SCOPE AND FOCUS OF THIS CHAPTER

Prior to accessing a data file, the accessing program must create a User File Block (UFB) for each file accessed, and supply the UFB with parameters that define the file. This chapter describes the UFB and its parameters, and presents an overview of the runtime operations involved in opening a disk file. It describes how you supply file definition parameters to the UFB, both at compile time and at runtime, and how these parameters are used by the OPEN statement. This chapter describes all user-defined file definition parameters used for disk files in RAM. Included in this chapter are details on:

- UFB parameters for RAM disk files
- The AXD1 for defining alternate indexed files using AXDGEN
- The OPEN macroinstruction
- The five file access modes used for opening disk files
- The CLOSE macroinstruction

While the descriptions of these file definition parameters are largely consistent for all VS languages, the syntax is different in the various high-level languages. Chapter 5 of this manual provides an overview of the DMS syntax of each of the VS high-level languages. High-level language programmers should use this chapter in conjunction with Chapter 5 and the reference manual for the specific high-level language.

You can supply some file definition parameters at runtime by workstation interaction or Procedure Language. These runtime file definition parameter assignments are the same for all languages. In Assembly language, you can supply file definition parameters directly to the UFB, or by using the UFBGEN and AXDGEN macroinstructions.

After you open a file, you access records in the file by issuing function requests. The five function requests (READ, WRITE, START, REWRITE, DELETE) are described in Chapter 7.

Information in this chapter is important for any type of DMS file definition, but the scope of this chapter is limited to the most common type of DMS file access: RAM access of disk files. Once familiar with the material in this chapter, users accessing files on devices other than disk should consult Chapter 11 for Workstation files, Chapter 12 for Magnetic Tape files, or Chapter 13 for Printer, Program and WP files.

6.2 THE USER FILE BLOCK

Every file accessed by a program must have a UFB to store the parameters that describe the file. There must be a separate UFB for every file that the program has open simultaneously. Two or more files may share a UFB if the program never has them open simultaneously. For files to share a UFB, you must supply different values to the permanent file name field at runtime, either through GETPARM interaction, or through a routine in the program that rewrites this field.

The UFB stores both file definition parameters for the OPEN statement, and function request modifier values. File definition parameters include the file organization and record type, the device type, the file, library, and volume names, buffering options, the block and the record lengths, and the primary key position and length. Many file definition parameters take default values if you do not specify a parameter value. The default value of a UFB parameter is always expressed as zeros. You must supply all necessary file definition parameters to the UFB before issuing an OPEN statement for the file.

In addition to the file description parameters, the UFB contains an internal flag that indicates that the file is open, and a second flag indicating in what mode the file has been opened. The UFB contains the addresses of the function request routines, and stores the name of the last function request executed and its modifier in UFBLF and UFBLFMOD. The execution of a function request returns a File Status code (FS), which indicates successful completion or the kind of error encountered. The file status fields UFBFS1 and UFBFS2 receive these numeric codes. Depending on the file status, the UFBERRAD or UFBEODAD error routine addresses may be taken. Further details on error routines may be found in Chapter 14.

When you open a file, DMS allocates an Open File Block (OFB) in Segment 0 (the system segment), and connects it to the UFB. DMS writes the UFB address in the OFB, and the OFB address in the UFB. It is through this linkage that the currently executing task is connected to the UFB. When the file is closed, DMS deletes the OFB address from the UFB.

Other control block addresses are also recorded in the UFB. The UFB does not contain parameters defining alternate key fields. It does, however, contain the address of the AXD1 block, which contains alternate indexed file information. Similarly, the UFB does not control buffer pooling directly but contains the Segment 2 address of the Buffer Control Table.

6.2.1 Creating the User File Block

When you assemble an Assembly language program, the assembler generates a UFB for each UFBGEN specified. The compiler automatically generates a UFB for a file defined in a high-level language. The UFB for each file is built as part of the object code module, and is automatically placed in the user's Segment 2 (program data) area when the program is loaded. The system retains these UFBs in Segment 2 for the duration of the program run.

The UFB is usually created in the Static portion of the user's Segment 2 by means of UFBGEN in Assembly language, or a similar data section paragraph (e.g., SELECT) in a high-level language. However, you can build the UFB directly in any portion of the user Segment 2, including the Stack area. See Chapter 15 for details concerning building the UFB directly. This chapter assumes the use of UFBGEN or one of its high-level language equivalents.

6.2.2 Addressing the User Record Area with UFBRECAREA

The user record area is a portion of the user Segment 2 of special concern for DMS RAM processing. It is a contiguous area of data storage the size of the largest permitted single record in a file. Program compilation (or assembly) creates at least one user record area for each file the program has open in RAM. You place the address of the user record area in UFB field UFBRECAREA.

A file can have more than one user record area in the user's Segment 2. If you wish to establish multiple record types (as described in Chapter 15), you should create several record areas in your Segment 2. In COBOL, you must use different record areas to set different alternate index bit map field values. You can modify the UFBRECAREA field while the file is open to change record areas.

You can only code program routines to change UFB parameter values in Assembly language. To change the value of a UFB parameter, you must first establish addressability to the file's UFB, as described later in this chapter. You should not attempt to change the values of most UFB fields while the file is open. Changing most UFB parameter values while the file is open produces unpredictable results. Those fields that are user-modifiable include:

- UFBRECAREA, the address label of the user record area.
- UFBKEYAREA, the address label of a field holding a relative record number (for relative or consecutive files), or a primary or alternate key value (for indexed files).
- UFBEODAD and UFBERRAD, user-defined error routine addresses. These fields are described further in Chapter 14.
- UFBGKSIZE, the generic key size used with START HOLD,RANGE or START EQ for locating or holding indexed records based on the initial characters of a key field.

When a file is closed, DMS restores the UFB parameter values so that a subsequent OPEN statement will access the same file. Therefore it is not necessary to respecify UFB parameters to reopen a file. To reuse a UFB for another file, you should reinitialize the UFB.

6.2.3 Creating the File Descriptor Record from the UFB

The Volume Table of Contents (VTOC) for a disk contains a File Descriptor Record (FDR) for each data file on that volume. An FDR consists of an FDR1 block, and an optional FDR2 block used to store data about additional file extents. When you create a file (by opening it in Output mode), DMS creates an FDR1 by copying parameter values from the UFB. The FDR1 permanently retains certain parameter values from the UFB, including file type, maximum record size (RECSIZE), flags for variable length and compression, the primary key position (KPOS) and length (KSIZE), and the packing densities (DPACK and IPACK) established for an indexed file, and the file name. The file, library, and volume names are retained on other VTOC blocks. These are permanent attributes of the file, and cannot be subsequently modified, except by a rename operation.

File state information, which is modified by the processing of a file, is also recorded in the FDR1. When you close a file, DMS automatically updates file state fields in the FDR1. These fields include the number of records in the file (NRECS), the relative block number of the last block in the file containing data (EBLK), and (for consecutive and relative files) the number of records in that end block (EREC). When a file is closed, DMS updates the NRECS, EBLK, and EREC fields of the FDR1.

To access an existing file, you must supply certain file identification parameters, including the permanent file, library, and volume names. These parameters allow DMS to locate the FDR record for the file. When you open the file, DMS copies parameter values from the FDR1 to the UFB. You can supply other parameters either through the program or through a workstation interaction (see Appendix D for details on supplying parameter values at runtime). If you do not supply a value for the library or volume parameters, DMS defaults to the usage constants of the person executing the program. You can set usage constants by pressing PF2 from the workstation Master Menu and typing in parameter values.

If you open an existing file without supplying a parameter value for one of the file's permanent attributes, DMS supplies the parameter value to the UFB from the FDR1. If you supply an incorrect value for one of the file's permanent attributes, the parameter value taken from the FDR1 usually overrides your parameter value.

This override feature insures that file formats remain consistent. However, because you may not be aware that a substitution has occurred, you may be performing DMS functions based on incorrect assumptions about the nature of the file being accessed. Prior to coding UFB parameters, or the user record area and key area fields for an existing file, you can use any of the following informational resources:

- The File Attributes screen for the data file. This screen is accessible from the Command Processor by using PF5. It supplies the record length (RECSIZE), the file and record structure (FORG, VLEN, and COMP), and the primary key length (KSIZE) and location (KPOS). This information should be used in coding the UFB parameters and in describing the user record area.

- The DISPLAY utility (optional). The Indices (PF3) option of DISPLAY supplies the number, position and length of alternate keys, and informs you whether duplicate alternate index key values are permitted.

- READFDR macroinstruction (optional), used to retrieve all file parameter values recorded in the FDR. Use READFDR to read UFB parameters values that are not available through the File Attributes screen or the Display utility.

6.3 UFBGEN

In Assembly language, you invoke the construction of a UFB in your static section by coding the file parameters in a UFBGEN macroinstruction. The UFBGEN shown in Example 6-1 creates a file with the UFB name "ZOOFILE", and supplies parameters describing the file organization, the record length, and the number of records.

Example 6-1. Generating a User File Block

```
         STATIC
ZOOFLE   UFBGEN PRNAME=Z001,FILENAME=ZOOFILE,LIBRARY=ZOOLIB,
                VOLSER=ZOOVOL,FORG=CONSEC,DEVCLASS=DISK,
                NRECS=100,RECSIZE=80,RECAREA=ZOOREC
         .
         .
         .
         UFB
         END
```

You should code a separate UFBGEN in the program's static section for every data file the program has open simultaneously. The UFBGEN macroinstruction does not produce executable code. Instead, during program assembly, the system uses the UFBGEN parameters to fill in a User File Block (UFB) with the specified fields initialized in Segment 2. Each UFBGEN creates a separate UFB. At the end of the STATIC section you should code a UFB statement, unless you have specified UFB NODSECT.

The names of the fields in the UFB begin with the three letters "UFB". You can instruct the assembler (or compiler) to insert a suffix letter as the fourth character of each field name; the rest of the field name generally corresponds to the name of the UFBGEN parameter.

6.3.1 Establishing References to the UFB

Some DMS operations require you to directly read or modify fields in the file's UFB. You can only perform operations of this type in Assembly language. If you wish to address fields in the UFB, you must first establish the addressability of the UFB. There are two methods of establishing UFB addressability.

Base Register Addressing

You can establish the UFB address by specifying a register to hold the address of the file's UFB. If you use a USING statement to equate this register with the base address of the UFB, you can access the fields of the UFB directly by name. However, you must change the base address whenever you change UFB addressing from one file to another. Therefore, this method is recommended when you are not frequently switching between multiple UFBs.

Example 6-2. Establishing UFB Addressing

```
        CODE
        LA      R6,ZOOUFB
        USING   UFB,R6
        .
        .
        .
        STATIC
ZOOUFB  UFBGEN
        .
        .
        .
        UFB
        END
```

In Example 6-2, you load the address of the UFBGEN into a register (R6 in the example), then establish that register as the base register. In the STATIC section you specify the UFBGEN and its parameters, and also specify UFB to provide the UFB DSECT.

To access a specific field in the UFB, you simply specify the name of that UFB field:

```
MVC   UFBGKSIZE,=X'02'
```

If you load the UFB address in a register, but do not specify a USING statement, you can access a specific field in the UFB as follows:

```
MVC   UFBGKSIZE-UFBBEGIN(1,r),=X'02'
```

You locate the UFB field by its offset from the beginning of the UFB. In this example, l is the length of the desired field; r is the register containing the address of the UFB.

Suffix Addressing

You can establish the address of a particular UFB by supplying a unique suffix character. When the assembler creates the UFB, it uses this suffix to uniquely identify each UFB field. It uniquely identifies UFB fields by placing the suffix character between the control block designator (UFB), and the name of the individual UFB field.

You supply two lines of code in the STATIC section that allow program references to the parameters established by UFBGEN. Once you have established these reference points, you can access UFB field values in the code section of the program.

Example 6-3. Establishing a UFB Suffix Character

```
       UFB  NODSECT,SUFFIX=Z
       ORG  UFBZBEGIN
ZOOUFB UFBGEN
```

The first line indicates that the UFB DSECT is not specified, but is generated from UFBGEN. The optional suffix field allows you to specify a suffix character to be included in all UFB field names for that file. The second line returns the offset pointer to the beginning of the UFB to allow the system to insert the parameter values supplied in UFBGEN. When specifying the beginning point (UFBBEGIN), you should include the suffix character in the field name, if you established a suffix in the UFB NODSECT line.

A suffix character uniquely identifies a field as belonging to a particular UFB. All references to UFB fields must include the suffix character:

```
MVC    UFBZGKSIZE,=X'02'
```

Because the suffix character uniquely identifies a particular UFB, you can use the UFBBEGIN field as a synonym for the UFBGEN label:

```
READ   UFB=UFBZBEGIN
```

6.3.2 UFBGEN Format

The UFBGEN is coded in the STATIC section of an Assembly language program, and consists of three parts: the label, opcode, and operands. The UFB label is a user-selected internal file name of eight characters or less, specified in the program for each OPEN, CLOSE, and function request operation on a file. It can be the same as the permanent file name, but does not have to be. The UFB label is specific to a particular accessing program; another program can access the same file using a different UFB label.

The opcode is UFBGEN, located in Column 7. The operands are the parameters described in greater detail in the following section. They are of the form:

parameter=value

separated from each other by a comma and containing no blanks. Parameters may be listed in any order, and the list may be extended to a second line by ending the first line with a comma, placing a non-blank character in Column 71, and continuing in Column 13 of the next line.

6.4 UFBGEN PARAMETERS

DMS requires several items of information to open a file in RAM (File definition parameter requirements for BAM and PAM are described in Chapter 10). You can provide these file description parameters in several different ways:

- You supply parameter values through UFBGEN within the static section of the program.
- You can supply parameter values by addressing the UFB directly, using an instruction in the code section of the Assembly language program or subroutine.
- You can accept the default values that DMS automatically supplies for some of the UFB file definition parameters at runtime.
- You can supply parameter values through GETPARM screens when the program is run interactively, or you can supply parameter values to GETPARMs through Procedure Language statements.
- DMS supplies some parameter values for existing files from data stored in the VTOC.

You must supply a set of file definition parameters to create a new file; it is not necessary to supply all of these file definition parameters to open an existing file.

When creating a file, DMS requires the record size (RECSIZE) and number of records (NREC) to assign space for the file; the file organization parameter (FORG) to determine how to construct the data file; and the file, library and volume names to determine where to store the file. You can supply the number of records, and the file, library and volume names through either UFBGEN as part of the program, or at runtime through GETPARM screen interaction or Procedure Language statements. The UFBGEN values, if specified, act as program defaults, enabling operator-free execution. These values can be overridden by values supplied to GETPARM at runtime through Procedure Language statements or workstation screen interaction.

It is not necessary to specify RECSIZE or FORG for existing data files. If you supply values for these parameters that disagree with the actual file values, DMS issues a respecification screen at runtime to respecify the file name.

The following section describes the UFB fields that you can define through UFBGEN parameters. UFBGEN parameters do not have to be defined in any particular order.

UFBGEN Parameters Required for All Data Files

RECAREA The address of the user record area, the area used to receive individual records for processing. You can supply this address as either an address expression of up to sixteen alphanumeric characters (the first character must be a letter and no blanks or special characters may be included), or a register number. The name you use to address RECAREA may be the same as the filename, but cannot be the same as the UFB address. You can modify the value of the RECAREA field while the file is open.

Example 6-4. Use of RECAREA

```
         READ  UFB=INFILE
MOVEREC  MVC   OUTREC,INREC
         WRITE UFB=OUTFLE
         .
         .
         .
         STATIC
INFILE   UFBGEN  RECAREA=INREC,FILENAME=OLDFILE
OUTFLE   UFBGEN  RECAREA=OUTREC,FILENAME=NEWFILE
```

FILENAME	The permanent name recorded in the VTOC and used in the system libraries to represent the file. This user-selected name is up to eight alphanumeric and/or national characters in length, with the first character representing the type of file. If it is a permanent file, you should begin the file name with a letter or number. If you do not include the FILENAME parameter in UFBGEN, the system issues a GETPARM at runtime requesting a valid file name. You can satisfy this GETPARM by either supplying a file name value in Procedure language, or typing the file name at the workstation screen. Refer to Appendix D. If you specify an @ character as the first character of the file name, and you have defined the file as a consecutive log file, DMS processes the file using automatic write-through. Write-through bypasses record buffering and writes each record directly to disk storage. Log files are described in Chapter 8. If you specify a file name of #, DMS creates a work file. If you specify a file name of ##, DMS creates a temporary file. When you specify # characters in place of a permanent file name (e.g., FILENAME=# or FILENAME=##), DMS automatically creates a file name for the file by taking the first four letters of name of the program creating the work or temporary file, followed by a four-digit number (e.g., EDIT0001). This number is incremented for every work or temporary file created during the period that the user is logged on. Work files are automatically scratched when they are closed. Temporary files are automatically scratched when an unlink operation exits the task, returning you to the Command Processor.
LIBRARY	The user-selected name of the VTOC library in which to place the data file. The same naming conventions apply as for file names. If you omit it, the system supplies the library name established as the usage constant for the current user of the program. You can set this parameter at runtime through a GETPARM screen interaction or through Procedure Language assignment. If the file is a work or temporary file, it is stored in library #xxxWORK, where xxx is the logon ID of the user. If a temporary file was created, this library is scratched when the user returns to the Command Processor.
VOLSER	The disk volume on which a data file is located, or the volume to which it should be written. VOLSER is a six-character name supplied either in UFBGEN or at runtime through GETPARM screen interaction or Procedure language assignment. If you do not code a value for this parameter, DMS supplies the usage constant volume for the current user of the program.

PRNAME	The parameter reference name is an internal file name of up to eight characters in length. You can specify any displayable characters for the PRNAME. Parameter reference names are used to identify the GETPARM screens used for runtime definition of file parameters and for writing procedures for these screens (see Appendix D). Specifying a parameter reference name rather than a permanent file name in UFBGEN allows you to assign file names at runtime, and to use the same UFB for access to several different files.

<u>UFBGEN Parameters Required for Creation of New Files</u>

FORG — The file organization. Specifies the structure of the file, either the structure to be created for a new file, or the structure for an existing file.

FORG=CONSEC	consecutive file organization.
FORG=REL	relative file organization.
FORG=INDEXED	indexed or alternate indexed file organization.
FORG=ANY	any organization acceptable. Used only for accessing existing files. The system supplies the correct file organization parameter value at runtime from the FDR block of the VTOC.

RECSIZE — The record size in bytes of the largest data record to be placed in the file. Maximum sizes for disk files range from 2024 to 2048 bytes, depending on the type of file (refer to Chapter 2). DMS uses this parameter value for allocating extents during file creation. For fixed length records, RECSIZE is the actual record length; for variable length and compressed files, set RECSIZE to the maximum uncompressed record length for the file. For relative files, RECSIZE represents the logical record size, not the record slot size. RECSIZE does <u>not</u> include the record length indicator or alternate index bit map suffix fields.

You supply the initial value for RECSIZE. If the file contains variable length records, DMS resets RECSIZE to the uncompressed record length of the current record as each record is processed. However, DMS preserves the initial maximum record length value (in UFBLRECSAVE), and reestablishes this RECSIZE value when you close the file.

You do not have to specify a RECSIZE value when you open an existing file. If you supply a RECSIZE larger than the actual maximum record size for an existing file containing variable length records, DMS opens the file using the original maximum record size. If you supply a RECSIZE smaller than the actual maximum record size for an existing file containing variable length records, DMS does not open the file, but displays a file respecification screen.

NRECS The number of records you expect to write to the file in Output mode. DMS uses this number for allocating extents during file creation. You can specify NRECS at runtime through a GETPARM screen interaction or Procedure Language statement.

When creating a relative file, you specify the desired number of record slots in the NRECS parameter. This number may be considerably larger than the actual number of records to be written to the relative file.

If you specify an insufficient number of records in NRECS, the number of records you write to the file may exceed both the space initially allocated to the file, and the number of additional extents that can be allocated in Output mode. In most cases, the total amount of allocatable space is approximately twice the number of records specified in NRECS. If your program exceeds the allocatable space, the file is automatically closed, and must be reopened in another mode to continue inputting records. For the effects of inaccurate approximation, see Chapter 3.

KEYAREA This optional parameter is used for random access to records in consecutive, relative, and indexed files. It takes an eight-character name (address expression) that addresses a data field. DMS uses the key values you place in the named data field to locate a particular record within a file.

In consecutive files, the KEYAREA addresses a four-byte area containing either a signed number indicating how many records to skip using START SKIP, or an unsigned relative record number used by READ REL.

Relative files use the four-byte area addressed by KEYAREA to hold the relative record number used for a WRITE, READ REL, REWRITE REL, or DELETE REL operation. You also use KEYAREA to address a relative record number area to locate a record in a relative file using a START EQ, GT, GE, LT, or LE operation.

When used for indexed and alternate indexed files, KEYAREA addresses a data field containing a primary or alternate key value that is used for READ KEYED, START EQ, START GT, or START GE. For files opened in Shared mode, KEYAREA must address a field within the record area.

In Example 6-5, the primary key (PRIKEY) is eight characters in length. The KEYAREA field, SEARCH, specifies a particular eight character primary key value.

Example 6-5. KEYAREA Parameter Coding for an Indexed File

```
        STATIC
ZOOFILE UFBGEN  FORG=INDEXED,RECAREA=ZOOREC,KEYAREA=SEARCH,
                KPOS=10,KSIZE=8
ZOOREC  DS      0CL80
FIELD1  DS      CL10
PRIKEY  DS      CL8
FIELD2  DS      CL65
         .
         .
SEARCH  DC      C'Aardvark'
```

<u>UFBGEN Parameters Required for Indexed Files</u>

KPOS	This parameter specifies the position of the primary key within the record. It is used only for indexed and alternate indexed files. You supply a primary key position as the number of bytes from the start of the data record, counting from zero. Do not count the record length indicator bytes.
KSIZE	This parameter specifies the length of the primary key within the record. It is used only for indexed and alternate indexed files. You supply the primary key length as the number of bytes in the key, counting the first byte of the key as one.
ALTCNT	The alternate index count parameter is used only for alternate indexed files. It specifies the number of alternate indices for the file (UFBALTCNT). Valid values are 0 to 16. The default value is zero. Specify ALTCNT in conjunction with ALTAREA. (See Section 6.4, AXDGEN.)
ALTAREA	Used only for alternate indexed files, the ALTAREA parameter specifies the address of the AXD1 block as generated by the AXDGEN macroinstruction. The AXD1 block's address is stored in User File Block field UFBALTPTR. Specify ALTAREA in conjunction with ALTCNT. (See Section 6.5, AXDGEN.)

Example 6-6. ALTAREA Coding for an Alternate Indexed File

```
ZOOREC  UFBGEN FORG=INDEXED,ALTCNT=2,ALTAREA=ZOOAXD
.
.
.
ZOOAXD  AXDGEN
```

UFBGEN Parameters for Record and File Types

VLEN — The variable length record parameter specifies whether the records in the file are to be formatted as variable length records. You must set VLEN equal to YES to create variable length or compressed records. For fixed length records, you can set this parameter equal to NO, or omit it; the default is fixed length records. Log files always contain variable length records.

COMP — The record compression parameter specifies whether all records in the file are to be compressed. You can set this parameter equal to YES to indicate that the file being created contains compressed records. If COMP is YES, VLEN should also be set equal to YES. If you set COMP=YES, but VLEN is not specified, DMS turns compression off. Print files always contain compressed records. Relative files cannot contain compressed records.

```
ZOOFILE UFBGEN  VLEN=YES,COMP=YES
```

PROG — The program file parameter specifies whether the file contains object code. The possible values are YES and NO, with NO as the default. Program files are described in Chapter 13.

PRINT — The print file parameter specifies whether the file is a print file. The possible values are YES and NO, with NO as the default. If you specify PRINT=YES, you must also specify VLEN=YES and COMP=YES. Print files are described in Chapter 13.

UFBGEN Special-Purpose or Informational Parameters

MODE — The mode parameter specifies the types of operations that you can perform on a file. You must specify a mode each time you open a file, either in the UFBGEN or in the OPEN statement. A mode specified in UFBGEN is the initial value for the file, which is overwritten by a mode specified in the first OPEN statement.

Possible values for MODE are: IN, OUT, IO, EXTEND, and SHARED. For a description of these values see the description of the OPEN statement MODE operand in this chapter. Input and Output should be abbreviated (IN and OUT) when coded in the UFBGEN.

FILECLAS	Specifies the file class of the file being operated upon (UFBFPCLASS). Valid values are A-Z, #, $, @, and blank. The default is 'blank'. You can specify the file class at runtime through GETPARM screen interaction or a Procedure Language statement.
NODISPLAY	Specifies that if you supply valid file definition parameter values before opening a file, DMS will neither display GETPARM screens to respecify these parameter values at runtime, nor accept parameter values from Procedure language routines. The NODISPLAY parameter is stored in the UFB as UFBF1NODISP. Valid NODISPLAY parameter values are YES and NO, with NO as the default. Refer to Appendix D for further details on runtime respecification via GETPARMS.
VERIFY	Requests read-after-write verification of a file (UFBF4VERIFY). Values are YES and NO. This option can significantly degrade performance, and its use is discouraged.

UFBGEN Parameters for Specific I/O Devices

DEVCLASS	Specifies the type of I/O device used for file access. The available values are: PRT (Printer), WS (Workstation), MTAPE (Magnetic Tape), and DISK. DISK is the default option. You can specify the device class at runtime through GETPARM screen interaction or a Procedure Language statement.
DEVNO	Identifies the device to be used by the program. Values must be integers. The system administrator establishes the device numbers for a VS configuration by running the GENEDIT utility. Device numbers differ among VS systems, with the exception of Device Number 0, which is always the operator's console workstation. You may have to respecify this field if the system is reconfigured, or if the program is run on another machine. This parameter is rarely specified.
PRTCLASS	Specifies the validation class to be used for a print file (UFBFORGPRINT) to specify on which device the file may be printed. Values can be A-Z. The default is A.

UFBGEN Parameters Used for DMS Error Processing

ERRAD	Specifies the address of the instruction to be branched to in the event of a file status code value of 30 or greater (UFBERRAD). ERRAD may be either a register specification or an expression. If ERRAD is left at its default value of zero, a fatal error occurs when the system returns a file status code of 30 or greater. See Chapter 14.

Example 6-7. ERRAD Coding

```
          CODE
          .
          .
          .
MSG       [error recovery subroutine]
          .
          .
          .
          STATIC
FILEA     UFBGEN  ERRAD=MSG
```

EODAD	Specifies the address of the instruction to be branched to in the event of any file status code from 10 through 29 (UFBEODAD). See Chapter 14.

UFBGEN Parameters Required for Packing Density

DPACK	Specifies the packing density for records in data blocks. You can only specify a packing density when creating an indexed or alternate indexed file. A packing density is expressed as the percentage of each data block that you can write to in Output mode. For example, a DPACK value of 60 would indicate that you can use 60% of the space in each data block to initially write records, with 40% of the block space reserved for the subsequent enlargement of the file. See Chapter 9 for further details.
IPACK	Specifies the packing density for entries in index blocks. This field only applies to primary index tree blocks; alternate index tree blocks are always packed at 100%. An IPACK value of 60 would indicate that when you create an indexed file, DMS initially uses 60% of each primary index block to store index entries and reserves 40% of the index block space for later enlargement. (See Chapter 9 for further details.)

UFBGEN Parameters Required for Buffering

BUFSIZE	Specifies the size in bytes of the Segment 2 buffer to be used for DMS processing of consecutive or relative files. Integer value must be a multiple of 2K bytes up to a maximum of 18K, or else the default (2K bytes) is used. Accepted values: 2048, 4096, 6144, 8192, 10240, 12288, 14336, 16384, 18432
POOL	Specifies that buffer pooling is to be used (UFBF1POOL). POOL must be specified in conjunction with BCT. Valid values for POOL are YES and NO. Use buffer pooling only when accessing existing indexed or alternate indexed files.
BCT	Addresses a Buffer Control Table in the user's Segment 2, as created by the BCTGEN macroinstruction (UFBBUFSTART). Specify an address expression of up to eight characters for the BCT parameter, as shown in Figure 6-8. BCT must be used in conjunction with POOL. It can only be used for indexed or alternate indexed files.

Example 6-8. BCT (Buffer Control Table) Parameter

```
FILEA   UFBGEN  FORG=INDEXED,POOL=YES,BCT=ZOOPOOL
.
.
.
ZOOPOOL BCTGEN
```

UFBGEN Parameters Used Only for BAM and PAM File Access

BAM PAM	When set equal to YES, these two parameters specify that you are accessing the file in either the Block Access Method (BAM) or the Physical Access Method (PAM). These two parameters are mutually exclusive. If neither is specified, DMS defaults to Record Access Method (RAM). See Chapter 10 for further details on BAM and PAM.
BLKSIZE	Specifies the block size (UFBBLKSIZE). The block size should always be 2048 bytes for disk files except when using PAM. (See Chapter 10 for PAM processing details.)
BLKAL	Valid only for BAM and PAM. Allocates space for a new disk file (UFBF4BLKAL) using the number of blocks specified in NBLKS. Valid values are YES and NO. You can allocate space in BAM or PAM either using BLKAL, or by specifying the number of logical records in NRECS. If you specify BLKAL=YES, you must also specify a value for NBLKS.

NBLKS	Valid only for BAM and PAM. Specifies the number of blocks to be allocated for a new disk file (UFBNBLKS). This parameter is required in conjunction with BLKAL.
NOVTOC	For diskette volumes only; identifies the volume as non-labelled, causing DMS to access the entire diskette as one file in BAM or PAM. DMS stores the NOVTOC parameter value in the UFB field UFBF4NOVTOC. Values are YES and NO, with NO as the default. When you specify NOVTOC=YES, DMS processing ignores RAM file definition parameters and the FILENAME, LIBRARY, and VOLSER parameters. NOVTOC diskettes do not support indexed files, or access to files in Shared or Extend mode.

6.5 AXDGEN

DMS generates an Alternate Index Descriptor block (AXD1) as Block 0 of every file accessible by alternate keys. The AXD1 block provides referencing material to the alternate key paths defined for the file. An indexed file can be defined with up to 16 alternate indices. To create an alternate indexed file, an Assembly language program should contain an AXDGEN for that data file. You should not use AXDGEN to access an existing alternate indexed file. Methods for accessing existing alternate indexed files are described later in this section. A complete Assembly language program example is provided in Appendix E.

NOTE

Files formatted for DMS/TX begin with two file recovery blocks. For these files, the AXD1 is in fact the third block of the file. However, these DMS/TX blocks are invisible to the Display utility and other user-accessible software. The AXD1 block is displayed as Block 0 for these files.

6.5.1 UFB Pointers to the AXD1

Every Assembly language program creating an alternate indexed file must have both an UFBGEN and an AXDGEN (or their equivalents). The UFBGEN should contain all the parameters required for an indexed file. These include FORG, KEYAREA, KPOS, and KSIZE. In addition, the file's UFBGEN should contain two special parameters, ALTAREA and ALTCNT, that are specific for alternate indexed files.

The ALTAREA parameter of UFBGEN supplies the location of the AXD1 block. The ALTCNT specifies the number of alternate index key paths specified for that data file. This number must be the same as the ENTRIES number in the AXDGEN, as shown in Example 6-9.

Example 6-9. The AXDGEN Macroinstruction

```
ZOOFILE UFBGEN  FORG=INDEXED,KEYAREA=PRIKEY,ALTAREA=ZOOAXD,
                ALTCNT=3
            .
            .
            .
ZOOAXD  AXDGEN  ENTRIES=3
```

6.5.2 Establishing References to the AXD1

Although a file's AXD1 is stored in block zero of the file, you cannot address the AXD1 directly in the file. Alternate indexed file processing copies the AXD1 into a buffer in the user's Segment 2. To perform an operation that requires you to directly read or modify fields in the file's AXD1, you must establish the address of the copy of the AXD1 in Segment 2. You can only perform operations of this type in Assembly language. There are two methods of establishing AXD1 addressability.

Base Register Addressing

You can establish the AXD1 address by specifying a register to hold the address of the file's AXD1. If you use a USING statement to equate this register with the base address of the AXD1, you can access the fields of the AXD1 directly by name. However, you must change the base address whenever you change AXD1 addressing from one alternate indexed file to another. Therefore, this method is recommended when you are not concurrently accessing multiple alternate indexed files.

Example 6-10. Establishing AXD1 Addressing

```
        CODE
        USING  AXD1,R6
        OPEN   UFB=ZOOUFB,MODE=IO
        L      R6,UFBALTPTR
        LA     R6,0(,R6)
        .
        .
        .
        STATIC
ZOOUFB  UFBGEN  FORG=INDEXED,FILENAME=ZOOFILE,LIBRARY=ZOOLIB,        X
                VOLSER=ZOOVOL,DEVCLASS=DISK
        .
        .
        .
        AXD1
        END
```

In Example 6-10, you establish a register as the base register for the AXD1. You then open the alternate indexed file, and establish the addressability of the AXD1 from the UFB. The load address (LA) instruction is used to clear the high-order byte of the register.

In the STATIC section you specify the UFBGEN; because you have already provided the UFB with the AXD1 address, you do not need to specify the ALTAREA or ALTCNT parameters in the UFBGEN for an existing file. You do have to specify an AXD1 statement in the STATIC section to provide the ADX1 DSECT.

To access a specific field in the AXD1, you simply specify the name of that AXD1 field:

```
          MVC   AXD1MASK,=X'02'
```

If you load the AXD1 address in a register, but do not specify a USING statement, you can access a specific field in the AXD1 as follows:

```
          MVC   AXD1MASK-AXD1BEGIN(1,r),=X'02'
```

You locate the AXD1 field by its offset from the beginning of the AXD1. In this example, 1 is the length of the desired field; r is the register containing the address of the AXD1.

Suffix Addressing

You can address AXD1 fields by supplying each file's AXD1 with a suffix character that uniquely identifies the each AXD1 field. You supply two lines of code in the STATIC section that allow program references to the parameters established by AXDGEN. Once you have established these reference points, you can access AXD1 field values in the code section of the program.

Example 6-11. Establishing an AXD1 Suffix Character

```
                AXD1       NODSECT,SUFFIX=Z
                ORG        AXD1ZBEGIN
      ZOOAXD    AXDGEN     ENTRIES=2
```

The first line indicates that the AXD1 DSECT is not specified, but is generated from AXDGEN. The optional suffix field allows you to specify a suffix character that DMS will include in all AXD1 field names for that file. The second line returns the offset pointer to the beginning of the AXD1 to allow the system to insert the parameter values supplied in AXDGEN. When specifying the beginning point (AXD1BEGIN), you should include the suffix character in the field name, if you established a suffix in the AXD1 NODSECT line.

A suffix character uniquely identifies a field as belonging to a particular AXD1. All references to AXD1 fields must include the suffix character:

```
         MVC    AXD1ZGKSIZE,=X'02'
```

6.5.3 Coding the AXDGEN

The label name for AXDGEN is the same as the value specified in the UFBGEN parameter ALTAREA. AXDGEN requires one parameter, ENTRIES=, and as many sets of subparameters as the value specified in ENTRIES=. You must code a set of three subparameters for each of the ENTRIES specified in the AXDGEN. Each set of subparameters specifies the options for one of the alternate key paths. The three subparameters are: ORD, which specifies which alternate key path is being referred to; KEYPOS, which specifies the beginning location of the alternate index key value in the records; and KEYSIZE, which specifies the length in bytes of the alternate index key value. These three subparameters are grouped together in parentheses, and separated by commas from the ENTRIES parameter and from other subparameter groups.

Example 6-12. AXDGEN Parameters and Subparameter Groups.

```
ZOOAXD  AXDGEN  ENTRIES=3,(ORD=1,KEYPOS=5,KEYSIZE=5),(ORD=2,       X
                KEYPOS=10,KEYSIZE=5),(ORD=3,KEYPOS=40,KEYSIZE=2)
```

AXDGEN Parameters

ENTRIES	The number, counting from one, of alternate index key paths established on the data file. This value must correspond to the number specified in the ALTCNT parameter of UFBGEN, and with the number of subparameter groups specified in AXDGEN. The highest possible value is 16. Keeping the number of alternate index key paths as low as possible enhances efficient addition and deletion of records. ENTRIES is a required AXDGEN parameter.
MASKAREA	The size in bytes of the PMASK area of the AXD1 block (see Chapter 3). At present, the only acceptable value for this parameter is 2. This parameter is optional.

AXDGEN Subparameters

Example 6-13. Use of AXDGEN Subparameters

```
ZOOFILE AXDGEN  MASKSIZE=2,ENTRIES=2,(ORD=1,KEYPOS=12,             X
                KEYSIZE=4,NODUPS),(ORD=2,KEYPOS=25,KEYSIZE=2)
```

ORD	The number of the alternate index path for a subparameter group. You should assign ORD values in ascending sequence from 1 to 16. ORD is a required subparameter.

KEYPOS	The position in the record of the beginning of the alternate index key value. This is the number of bytes from the beginning of the uncompressed record, counting from zero. This count excludes the record length and block length indicator bytes. KEYPOS is a required subparameter.
KEYSIZE	The length in bytes of the alternate index key for that key path. Make alternate index keys as short as possible to save space and improve performance. KEYSIZE is a required subparameter.
NODUPS	Specifies whether duplicate alternate key values are to be allowed for a specified alternate key field. Whether or not you permit duplicate alternate key values when you create the file determines the size and structure of the alternate index key tree (see Chapter 3.4). Specify NODUPS without a value to prohibit use of duplicate alternate key values for the alternate key field. If you specify NODUPS, DMS will reject the writing or rewriting of a record that contains a duplicate value for that alternate key field. NODUPS is an optional subparameter.
COMPRESS	Not currently operational. COMPRESS is an optional subparameter.

6.5.4 Accessing the AXD1 of an Existing File

If you know how many alternate key entries are in an AXD1, and you are not opening the AXD1 in Shared mode, you can allocate space for the AXD1 in your Segment 2 space, as shown in Example 6-14.

Example 6-14. Defining Space for the AXD1 of an Existing File

```
         STATIC
         AXD1
ZOOFILE  UFBGEN  FORG=INDEXED,KEYAREA=PRIKEY,ALTAREA=ZOOAXD,
                 ALTCNT=3
ZOOAXD   DS   0F
         DS   (AXD1ENTRY-AXD1BEGIN+3*AXD1ENTRYLENGTH)X
```

This program defines AXD1 space by determining the length of the AXD1 header, then adding the length of each existing alternate index path entry, multiplied by the number of entries as specified in the UFBGEN ALTCNT field. When you open the file, DMS establishes the parameter values for this Segment 2 AXD1 from the parameter values in the file's AXD1 block.

You can open an existing alternate indexed file, and then establish addressing to the open file's AXD1 block. You must use this method to access the AXD1 if you are opening the alternate indexed file in Shared mode. You establish the address of the file's AXD1 as shown in Example 6-15.

Example 6-15. Establishing the AXD1 Address of an Existing File

```
CODE
OPEN  UFB=ZOOFILE
L     R6,UFBALTCNT
LA    R6,0(,R6)
USING AXD1,R6
.
.
.
MVC   AXD1MASK(2,R6),ZOOMASK
```

You must first open the alternate indexed file. You then load the number of entries specified in the UFB into a register. Given the number of entries, DMS can establish addressability to the AXD1. Once you have established addressability to the AXD1, you can modify its parameter values using move operations. The above example moves the value of ZOOMASK into the two-byte AXD1MASK field in the file's AXD1.

6.5.5 Accessing the Record Mask Bytes

When DMS places a record in an alternate indexed file, it adds to the end of each record a blank two-byte bit mask suffix. You must set these sixteen bits to indicate which of the sixteen possible alternate key paths can reference the record (see Chapter 3.4).

When creating a file, the appropriate value of the mask for each record should be moved to the AXD1 mask area. A simple move statement will perform this task. You should remember what suffix character (if any) you specified in the AXD1 NODSECT statement when specifying the AXD1 mask area.

Example 6-16. Coding the Alternate Indexed Record Mask Bits

```
.        CODE
         OPEN
         READ
         MVC   AXD1MASK,ZOOMASK
         WRITE
         CLOSE

         STATIC
         UFBGEN
         AXD1GEN

         ZOOMASK DC  X'8000'
```

You can set mask values in hexadecimal or in binary, but in either case, you should establish the paths in sequence from left to right (hex values '8000', 'C000', 'C200', etc.). The value you establish in the record's bit mask suffix must be a subset of the number of paths specified in the ENTRIES parameter of the AXDGEN.

6.6 THE OPEN MACROINSTRUCTION

In order to create or access a data file, you must first open that file. This is done by coding an OPEN statement in the code section of the program. Every file to be opened must have a UFB. In Assembly language, you usually establish a UFBGEN in the static section of the program. An OPEN statement identifies the file to be opened by giving the address of the UFB (the UFBGEN label) as shown in Example 6-17.

Example 6-17. The OPEN Macroinstruction

```
        CODE
        OPEN         UFB=ZOOFILE,MODE=INPUT

        STATIC
ZOOFILE UFBGEN
```

The OPEN statement sets the AXD1ALTINX field to zero so that subsequent record access is by primary key. It also resets the current record pointer to the first record in the file.

6.6.1 Open Macroinstruction Syntax:

```
OPEN   UFB={(register)}   [,MODE= {OUTPUT}] [,{NOGETPARM}] [,EXIT={(register)}]
           {expression}           {INPUT}     {NODISPLAY}          {expression}
                                  {IO}
                                  {EXTEND}
                                  {SHARED}
```

6.6.2 File Access Modes

Every file is opened in a particular access mode. Generally, you state the open mode as part of the OPEN statement, as shown in Example 6-17. If you do not specify a mode in the OPEN statement, the mode specified in the UFB is used. This can be a mode you established using UFBGEN, or the mode you specified in the previous OPEN statement for that file. The mode specified in the current OPEN statement takes precedence over the mode specified in the UFB. The mode specified in the OPEN statement may, in turn, be overridden by a mode specified in a START OUTPUT, START EXTEND, or START IO function request. See Chapter 7 for additional information on the START function request.

DMS provides five file access modes for RAM disk files. Each of them has a specialized function, and allows a different subset of the user function requests. The five modes are as follows:

OUTPUT	Output mode for file creation
INPUT	Input mode for reading records from a file
IO	Input/Output mode for updating a file
EXTEND	Extend mode for expansion of a consecutive or relative file
SHARED	Shared mode for sharing a file

Output Mode

The Output mode is used for the initial creation of a file. The only function request processing that DMS supports in Output mode is the sequential writing of records into a file using the WRITE function request. Output mode can be used for consecutive, relative, or indexed files. You must supply indexed records to Output mode in ascending primary key sequence.

CAUTION

You can open an existing consecutive or relative file in Output mode. When you open an existing file in Output mode, DMS displays a screen warning you that to create a file with the specified file name and library, DMS must scratch a preexisting file. If you press PF3 from the warning screen, DMS scratches the old file, deletes the VTOC entry for the file, and deallocates file extents. The Open in Output mode uses the old file's UFB to create a new FDR1 entry in the VTOC, and to assign space to the new file. After opening this file, you can perform write function requests by writing records sequentially, beginning with Relative Record 1.

If a program crashes while an alternate indexed file is open in Output mode, the file will not be accessible by alternate index keys. DMS does not construct alternate index trees in Output mode until the file is closed. If the system crashes while a file is open in Output mode, records written to the output file are not preserved.

You can create a file by opening it in Output mode and then write zero records to it. This use of Output mode to establish file space is useful when, for example, indexed records used to create a file cannot be put in primary key order. Once you have established the file space in Output mode, you can write the records to the file in I/O mode.

Example 6-18. Use of the Output Mode

```
        L       4,COUNT
        OPEN    UFB=OLDFILE,MODE=INPUT
        OPEN    UFB=NEWFILE,MODE=OUTPUT
RETURN  READ    UFB=OLDFILE
        MVC     NEWREC,OLDREC
        WRITE   UFB=NEWFILE
        BCT     4,RETURN
        CLOSE   UFB=NEWFILE
        CLOSE   UFB=OLDFILE
```

As part of the Open operation in Output mode, DMS establishes a release option for the file. A release deallocates all unused blocks at the end of a file, making them available as a free extent to other files. Unused space is released as part of the Close operation. The release option default is to release unused space for consecutive files, and to not release unused space for indexed files. DMS cannot release unused space from indexed files containing alternate indices, nor can it release empty blocks embedded within the file data.

You can override the release option defaults by means of the GETPARM Screen for Output File Definition, shown in Appendix D. The displayed RELEASE option on this screen is the default for the file type. If you change the RELEASE option default, either through workstation interaction or Procedure language commands, DMS uses the option you specify. However, if you specify NODISPLAY=YES in the UFB, DMS automatically uses the default value for the release option.

After opening a file in Output mode, you can change the release option by modifying a field in the UFB. You can specify release by setting the UFBF4RLSE bit in the file's User File Block, as shown in Example 6-19.

Example 6-19. Releasing Unused Blocks

```
OPEN    UFB=NEWFILE,MODE=OUTPUT
OI      UFBF4,UFBF4RLSE
.
.
WRITE   UFB=NEWFILE
.
.
CLOSE   UFB=NEWFILE
```

To perform a release, you must set the UFBF4RLSE after opening the file, since the Open operation resets UFBF4RLSE to the system default. You can only release space from files in Output mode. You cannot release space from alternate indexed files.

<u>Input Mode</u>

The Input mode is used for reading records from a data file. You cannot modify a file that you have open in Input mode. You cannot write records to a file you have open in Input mode.

Because you cannot modify a file opened in Input mode, multiple users can concurrently read the same record in a file without waiting for another user to relinquish it. Records in DMS files are available for Input mode access regardless of concurrent Input mode access to those files by other users. You cannot, however, open a file in Input mode if that file is already open in another mode. You should open files accessed by many users in Input mode whenever possible.

<u>I/O Mode</u>

The Input/Output mode allows you to update a file by reading, modifying, and rewriting records in a data file. You can read and rewrite records in consecutive disk files, providing the record length remains constant. You can read, rewrite, delete, and add records to relative or indexed files in I/O mode.

Unlike Input mode, I/O mode restricts other users' access to the file. If multiple users need to simultaneously update records in a file, the file should be opened in Shared mode. The I/O and Shared modes provide the same functionality, except that Shared mode allows several users to simultaneously update the same file.

You can read records in I/O mode by using the READ HOLD function request, and return updated records to the file by issuing a REWRITE function request. (See Chapter 7 for further details on these function requests.)

Extend Mode

Use the Extend mode to enlarge an existing consecutive or relative file by adding records to the end of the file. When you open a file in Extend mode, DMS automatically locates the end of the data file, so that a WRITE function request places a record at the end of the file. Extend mode is used for consecutive and relative files; you cannot open an indexed file in Extend mode.

Shared Mode

Shared mode provides you with all of the functions of I/O mode. In addition, it allows more than one user to concurrently update a file. When you wish to update a file in a multiprogramming environment, you can open the file in Shared mode to avoid "locking out" other users. A time-out feature prevents contention for the same resource.

You can open indexed files in Shared mode for I/O update (adding, deleting, and modifying records). A consecutive file can be shared in either of two ways. You can open a consecutive disk file for I/O update in Shared mode. You can also designate a consecutive file as a log file. Several users can concurrently extend a log file by opening it as a shared file. Log file users add records to the end of the consecutive file in the order that the system processes the added records. This function is valuable for keeping an audit trail of activity in strictly chronological order. See Chapter 8 for further details on log files and I/O processing in Shared mode.

6.6.3 Other OPEN Macroinstruction Operands

NOGETPARM Suppresses runtime user interaction and causes procedure-supplied parameters to be ignored. NOGETPARM causes a GETPARM Type RD to be issued rather than a Type I. A Type RD GETPARM solicits no information from the workstation or from the procedure. You should only use this option if your program supplies all of the required runtime parameters. Along with NOGETPARM, you should also code Open Exits in your program to enable it to handle error conditions. NOGETPARM and NODISPLAY are mutually exclusive.

NODISPLAY Suppresses user interaction if the values supplied in the UFB or through Procedure Language statements are valid parameter values. NODISPLAY causes a GETPARM Type ID to be issued rather than a Type I. A Type ID GETPARM only solicits information from a procedure. NOGETPARM and NODISPLAY are mutually exclusive.

NOTE

User interaction will occur even if NOGETPARM or NODISPLAY is specified if a field contains a semantic error (e.g., if you specified an invalid device type).

EXIT You can specify the optional EXIT operand to indicate which file assignment problems should cause the system to return control to the issuing program rather than to display a respecification screen. If the Open operation fails, DMS checks the bit mask of the EXIT operand. If a bit is set in the EXIT operand that corresponds to the condition that caused Open to fail, DMS returns control to the program at the next instruction. You can specify an EXIT bit mask value as a register or as an absolute expression.

DMS indicates the failure of an Open operation by writing a file status character of '9' to UFBFS1. If you specified an EXIT operand, it indicates the type of Open failure in UFBFS2. If no EXIT operand is specified, or if the the EXIT operand bit is not set for a particular Open failure, DMS attempts to display a file respecification screen. A respecification screen is not displayed if you specified the NOGETPARM operand.

Example 6-20 shows one method of using the EXIT operand:

Example 6-20. Use of the OPEN EXIT

```
OPEN  UFB=ZOOFILE,EXIT=X'FF'
CLI   UFBFS1,UFBFS1SUCCESS
BNE   [error routine]
```

The above example sets all of the bits in the EXIT operand bit map by specifying a value of 'FF' to the mask. Therefore, any Open failure will change the value of the file status field UFBFS1. Following an Open Exit, the next program instruction is performed, in this case a test for the value of UFBFS1. If UFBFS1 contains a file status other than the status for successful completion, the program branches to a user-written Open failure routine.

You can specify any combination of the EXIT mask bits listed in Table 6-1.

Table 6-1. Open Exit Bit Mask Values

EXIT Mask Bit Set	Corresponding UFBFS2 Equate Statement	Open Failure Type
1000 0000	UFBFS2XFILE	Return if the file is not found (non-Output mode) or if a duplicate file name is found (Output mode).
0100 0000	UFBFS2XLIB	Return if the library is not found (non-Output mode).
0010 0000	UFBFS2XVOL	Return if the volume is not mounted.
0001 0000	UFBFS2XSPACE	Return if there is insufficient space on the volume for a new file (Output mode).
0000 1000	UFBFS2XVTOC	Return if there is no VTOC space on the volume (Output mode).
	UFBS2XTAPELD	Return if the tape label type or tape density is not acceptable to the program.
0000 0100	UFBFS2XPOS	Return for possession conflict. Possession conflict includes file already open by current program, file opened by other program and open modes conflict and volume possession is exclusive for another user. Error further described in UFBXCODE.
0000 0010	UFBFS2XPROT	Return if the user does not have access rights required to open the file.
0000 0001	UFBFS2XFORMAT	Return if there is an error in specification of file format. Error class is described in UFBXCODE.

Further information on interpreting file status values for Open Exits is provided in Chapter 14, DMS Error Routine Processing.

6.7 THE CLOSE MACROINSTRUCTION

You should close every file that you open. You perform this task by issuing a CLOSE macroinstruction after you finish record processing. You can open and close a file several times in a single program. When closing disk files, the CLOSE macroinstruction requires only one operand, the UFB address operand that specifies which file is being closed. Other parameters are available for closing tape files. Use of CLOSE for disk files is shown in Example 6-21.

Example 6-21. The CLOSE Macroinstruction

```
        CODE
        OPEN  UFB=ZOOFILE,MODE=INPUT
        .
        .
        .
        CLOSE UFB=ZOOFILE
        STATIC
ZOOFILE UFBGEN    FILENAME=MYZOO
```

The CLOSE macroinstruction frees system resources. Closing a file deallocates the OFB allocated to the file, allowing another task to use this OFB. Closing a file frees buffer blocks allocated in the user's Segment 2 area. You should close all files as soon as you have completed all processing for those files. However, if the file is attached to a DMS/TX database, issuing a Close statement performs a FREE ALL, affecting the processing of other files. Proper placement of CLOSE statements is critical for DMS/TX file processing; you should never close a file within a routine that performs processing of records in other files.

If the release option has been specified, either as the system default, through GETPARM interaction, or through user modification of the UFB, DMS releases all unused blocks of space following the last block of data in the file. The description of Output mode in this chapter provides more details on the release option.

The CLOSE instruction updates the FDR entry for the file, modifying the NRECS, EBLK, and EREC values. The CLOSE instruction also resets the UFBRECSIZE parameter value, using the value stored in UFBLRECSAVE.

In Output mode, DMS does not construct alternate index trees until a file is closed. During Output mode processing, DMS stores the key values used to construct the alternate key trees as temporary work records in the file. CLOSE initiates the construction of alternate index trees by linking to the BUILDALT system utility. BUILDALT sorts the temporary work records and formats them as alternate index tree records.

When creating a file, DMS can only identify illegal duplicate alternate key values after the file has been closed. If you input records with duplicate alternate key values for an alternate key path that disallows duplicate values, DMS only writes one record with that alternate key value to the file. It writes the record with the lowest primary key value; all other records with that duplicate alternate key value are written to an error log.

The system automatically closes all files left open when your program exits the link level in which the file was opened. All files left open are automatically closed at program completion. Any error that prevents an Output file from being properly closed results in a file with no usable alternate index trees.

CHAPTER 7
ACCESSING DMS DISK RECORDS USING FUNCTION REQUESTS

7.1 INTRODUCTION TO FUNCTION REQUESTS

You access DMS files using function requests. A function request is a runtime operation you issue to an open file. DMS supports five function requests: READ, WRITE, REWRITE, DELETE, and START. Using these function requests, you can locate individual records by means of a pointer, and read, write, or delete records. This chapter describes the function requests you use for RAM access of disk (or diskette) files, with terms and examples from Assembly language. However, most of the material in this chapter is applicable to all forms of DMS file access.

The function requests described in this chapter are as follows:

READ	Read a record from a data file.
WRITE	Write a new record to a data file.
REWRITE	Update an existing record in a data file.
START	Locate a record in a file, hold or release a record or group of records, or change the file processing mode.
DELETE	Delete a record from a relative or indexed file.

RAM function requests can read, write, and delete data records only; the AXD1, index blocks, DMS/TX file recovery blocks, and other non-data file blocks are transparent to RAM function request processing.

Assembly language provides a macroinstruction for each function request, with multiple modifiers for READ and START operations. High-level languages provide similar support, except for certain START operations that can only be performed in Assembly language. Each function request must include an operand specifying the address of the User File Block (UFB). You can specify as a value for this operand the name of the UFBGEN label (as shown throughout this chapter), or as the UFBBEGIN field of a UFB that you defined with a suffix value:

```
READ UFB=UFBZBEGIN
```

Assembly language supports conditional execution of function requests.

7.2 THE USE OF FUNCTION REQUESTS

You must open a file before you can perform a function request to access the data in the file. The mode that you specify when you open the file limits the available function requests to those that are appropriate for that mode of access. The Open operation and mode selection are described in Chapter 6. The data access method (e.g., RAM, BAM, PAM), the type of storage media, and the type of file also determine which function requests are available and how those requests are interpreted by the system.

Function requests are calls to DMS modules. You code them in the non-modifiable code section of a program according to the conventions of the programming language. Function requests can be (and usually are) mixed with the ordinary program instructions for the language. They must specify, using a UFB address, which file they are operating on. The corresponding file must be represented by a UFB in storage. Usually, you place this UFB in the static section of the program (e.g., via UFBGEN), but it is possible to place the UFB in heap storage or on the stack.

Example 7-1. Use of Function Requests in Assembly Language

```
CODE
OPEN     UFB=OLDFILE,MODE=OUTPUT
OPEN     UFB=NEWFILE,MODE=INPUT
READ     UFB=OLDFILE
MVC      NEWREC,OLDREC
WRITE    UFB=NEWFILE
CLOSE    UFB=OLDFILE
CLOSE    UFB=NEWFILE
```

The operation a function request performs is often dependent on runtime parameters established after the file is opened. For example, the READ NEXT, READ KEYED, and START EQ, GT or GR function requests normally locate records by primary key values. If, however, you set the AXD1ALTINX field before issuing the function request, these function requests locate records by alternate key values along the specified alternate key path.

When you perform a function request, DMS updates a pointer to your current record position in the file. If you perform an operation that reads the next record, for example, DMS determines what record to access based on this pointer. For consecutive files, DMS maintains the current record pointer value in the User File Block UFBLOGRECCNT field. For relative files, DMS maintains the current record pointer value in UFBRELPOS. These fields are read-only indicator fields; you cannot change file currency by changing these UFB parameter values. Unpredictable results and file damage may occur if you change the current record pointer value.

When you open a file, DMS establishes the current record pointer at the first record in the file. When you open a file in Extend mode, DMS establishes the current record pointer at the last record in the file that contains actual data. In consecutive and indexed files, every successful function request updates the current record pointer. In relative files, only successful READ and START function requests update the current record pointer.

7.3 THE READ FUNCTION REQUEST

In RAM, the READ function request retrieves a record from a file currently open in Input, I/O, or Shared mode. The result obtained from a READ depends on the value of its modifier. For instance, you must use a READ function request with a HOLD modifier to read records that are to be subsequently rewritten or deleted. READ function request modifiers are not always applicable to all types of files. The valid modifiers for various file types are shown in Table 7-1.

Table 7-1. Valid READ Modifiers for RAM Disk File Types

Fixed Length Consecutive Files	Variable Length Consecutive Files	Relative Files	Indexed Files
no modifier	no modifier	no modifier	no modifier
NEXT	NEXT	NEXT	NEXT
HOLD	HOLD	HOLD	HOLD
(HOLD,NODATA)	(HOLD,NODATA)	(HOLD,NODATA)	(HOLD,NODATA)
REL	NODATA	REL	KEYED
(REL,HOLD)		(REL,HOLD)	(KEYED,HOLD)
(REL,NODATA)		(REL,NODATA)	(KEYED,NODATA)
(REL,HOLD,NODATA)		(REL,HOLD, NODATA)	(KEYED,HOLD, NODATA)
NODATA		NODATA	NODATA

You must open a file in Input, I/O, or Shared mode, or place the file in temporary I/O mode using the START IO function, before invoking a READ operation. When DMS reads a record from a file, it copies the record data into the user record area (addressed by UFBRECAREA), unless you specify the NODATA modifier. DMS expands a compressed record as it copies it into the user record area. Other program instructions can subsequently process a record once it is in the user record area.

When you read a file containing variable length records, DMS indicates the length of the record currently being read in the RECSIZE field of the file's UFB. If the record belongs to an alternate indexed file, DMS copies the record's bit map suffix into the AXD1MASK field of the file's AXD1.

READ with no modifier and READ NEXT are functionally identical. You use these function requests to invoke a sequential read of records in the file. READ REL (Read Relative) and READ KEYED are used to locate and read specific records randomly within a file. Use READ REL for relative and consecutive files, READ KEYED for indexed files. To use either of these function requests, you must first establish a KEYAREA field to supply either the relative record number or the record key value.

You can use READ NEXT or READ KEYED to read records in indexed files by either primary or alternate index key. READ NEXT reads records either sequentially by ascending primary key value, or sequentially along the specified alternate key path. A plain READ KEYED locates records randomly by primary key value in either indexed or alternate indexed files. If a statement defining the value of the AXD1ALTINX field precedes the READ KEYED statement, it reads records randomly by alternate key value. A complete Assembly language program that accesses records by alternate key value is provided in Appendix E.

Multiple READ modifiers are necessary for certain access modes. For example, to randomly read a indexed file record that you plan to update you issue a READ (KEYED,HOLD). You may use the KEYED, REL, HOLD, and NODATA modifiers to construct multiple modifiers. The elements of a multiple modifier appear in parentheses, separated by commas, with no blanks. Modifiers in a multiple modifier may be in any sequence. The user program must fulfill all the conditions and restrictions for each element of a multiple modifier.

7.3.1 READ Function Request Syntax: RAM Disk Files

```
[label]  READ  [        ,]          UFB={(register)}    [,COND=number]
                NEXT                     {expression}
                HOLD*
                REL*
                KEYED*
                NODATA*
```

(May be used as an element of a multiple modifier. See Table 7-1.)*

7.3.2 READ Function Request Modifiers

The following is a list of the READ function request modifiers.

Modifier	Comments
NEXT or no modifier	READ with no modifier and READ NEXT are functionally identical. They place the next data record in the file into the user record area (addressed by UFBRECAREA). DMS identifies the next record using the value of the current record pointer. READ NEXT can be used on consecutive, relative, indexed, or alternate indexed files. In a relative file, a READ NEXT always returns the next logical record (record containing data). Empty record slots are passed over. When following OPEN, READ NEXT yields the first record in the file. When following a READ REL or a READ KEYED for a primary key, READ NEXT yields the next sequential data record in the file. When following a READ KEYED for an alternate key, a READ NEXT yields the next record on that alternate key path. When following a START statement that sets the current record pointer, a READ NEXT statement reads the record specified by the START function request. If you perform a READ KEYED or START that begins accessing records by an alternate key, a subsequent READ NEXT function request reads the next sequential record along that alternate index path. Subsequent READ NEXT function requests read along that key path in ascending sequence by alternate key value. Issuing a READ NEXT instruction can generate a File Status '10', indicating an end of file condition. File Status '10' indicates that either the physical end of the file has been encountered, or the last record has been read along that alternate key path. When reading along an alternate key path, DMS returns a File Status '02' if the file contains at least one more duplicate value for that alternate key; DMS returns a File Status '00' if all duplicate values for that alternate key have been read.
HOLD	Use READ HOLD to update a record in I/O or Shared modes. HOLD indicates that DMS retains the record read from the file in a user buffer until it is either rewritten, deleted, or released by another READ HOLD operation. You must specify a READ HOLD as a precondition for a REWRITE or DELETE of a consecutive or indexed file record. You must specify READ HOLD if you wish to read, and then rewrite or delete a record in a relative file; however, it is possible to rewrite or delete a relative file record without previously reading the record.

You can specify HOLD as an element of a multiple modifier. For a file opened in Shared mode, READ HOLD indicates that the record read will not be made available to any other user until you either rewrite the record, delete it, or issue a READ HOLD for another record in any shared file. Note that this means that a program may only HOLD one record at a time, no matter how many files are being shared. However, you can simultaneously hold multiple records in files attached to a DMS/TX database. See the VS DMS/TX Reference for further details.

Example 7-2. The READ HOLD Function Request

```
OPEN      UFB=ZOOFILE,MODE=IO
READ HOLD,UFB=ZOOFILE
REWRITE   UFB=ZOOFILE
```

REL

Used for relative files and fixed length consecutive files, READ REL locates and reads a record by its relative record number (from 1). You specify the record to be read in the four-byte data area in unsigned binary format addressed by the UFB field KEYAREA. See START SKIP for variable length consecutive files.

In a relative file, the Relative Record Number addressed by UFBKEYAREA must refer to a slot containing actual record data (zero-length records included). If you specify a Relative Record Number that refers to an empty record slot, DMS returns a File Status '23'.

If the KEYAREA value is zero, a negative number, or a value greater than the number of records in the file, DMS returns a File Status '23'.

You can code REL as an element of a multiple modifier with HOLD and/or NODATA.

Example 7-3. The READ REL Function Request

```
                   CODE
                   OPEN UFB=ZOOFILE,MODE=INPUT
                   MVC  ZOOKEY,=F'10'
                   READ REL,UFB=ZOOFILE
                   .
                   .
                   .
          STATSEC  STATIC
          ZOOFILE  UFBGEN FORG=CONSEC,KEYAREA=ZOOKEY
                   .
                   .
          ZOOKEY   DS F
```

KEYED [by primary key]

Use READ KEYED to retrieve records by primary key value in indexed and alternate indexed files. Specify the primary key value beginning at the address in the UFB field KEYAREA, and extending for the number of bytes specified as the primary key size (UFBKEYSIZE). The area addressed by KEYAREA should be in the proper location within the file's record area.

If the READ KEYED value does not correspond to an existing key value, DMS returns a File Status '23'.

You can code a READ KEYED as an element of a multiple modifier with HOLD and/or NODATA.

If you specify a READ with no modifiers after a READ KEYED, DMS reads the next record in primary key sequence after the keyed record.

Example 7-4. The READ KEYED Function Request for a Primary Key

```
                   CODE
                   OPEN  UFB=ZOOFILE,MODE=INPUT
                   MVC   PKAREA,PKVALUE
                   READ KEYED,UFB=ZOOFILE
                   .
                   .
                   .
          STATSEC  STATIC
          ZOOFILE  UFBGEN FORG=INDEXED,RECAREA=ZOOREC,                X
                          KEYAREA=PKAREA,KSIZE=5,KPOS=0
          ZOOREC   DS 0CL80
          PKAREA   DS CL5
                   DS CL75
          PKVALUE  DC C'00017'
```

KEYED [by alternate key]

READ KEYED can also be used to read alternate index records randomly by alternate index key value. After opening the file, you must pass the alternate index ordinal parameter to the AXD1, establishing the alternate path for record access. This alternate index ordinal field, AXD1ALTINX, holds a single binary byte, with possible values of 0 to 16. Access by primary key is the 0 (default) value; values from 1 to 16 specify alternate paths. When you open a file, DMS initializes AXD1ALTINX to 0. Performing a WRITE operation resets AXD1ALTINX to 0.

After setting AXD1ALTINX, move an alternate key value to the field addressed by KEYAREA, and issue a READ KEYED function request. The area addressed by KEYAREA should be in the proper location within the file's record area. If duplicate alternate key values exist, DMS retrieves the first record (lowest primary key) with that alternate key value. Other records with the same alternate key value can be read by successive READ NEXT operations. If no alternate key exists with the specified value, DMS returns a File Status '23'.

The following partial program performs an indexed read by alternate key value for a file with three alternate key paths. It searches Alternate Key Path 2 for a record with an alternate key value of 0057.

Example 7-5. The READ KEYED Function Request for an Alternate Key

```
         CODE
         OPEN    UFB=ALTFILE
         MVC     KEYA,ALTVAL
         MVC     AXD1ALTINX,PATHNO
         READ KEYED,UFB=ALTFILE
         .
         .
         .
         STATIC
ALTFILE  UFBGEN FORG=INDEXED,KEYAREA=KEYA,                     X
                ALTAREA=ZOOALT,ALTCNT=3
         AXD1   NODSECT
         ORG    AXD1BEGIN
ZOOALT   AXDGEN  ENTRIES=3,
ALTVAL   DC     C'0057'
PATHNO   DC     X'2'
```

NODATA	A READ NODATA places the requested file data in the buffer area, but does not copy the specified record into the user record area. Instead, the record is retained in the DMS buffer, and the address of the record in the buffer is placed in Register 1. If the file is compressed, the record stored in the buffer is in compressed format. A READ NODATA does not update UFBRECSIZE. READ NODATA can be used with consecutive, relative, or indexed files. You cannot use READ NODATA in Shared mode, or for processing compressed records. NODATA can be used by itself, or as part of a multiple modifier with KEYED, REL, or HOLD.

7.3.3 READ Function Request Operands

Operands	Comments
UFB	This operand is required for all READ statements. It specifies the address of a User File Block. You can supply it either as a register specification (where the register contains the UFB address), or as an address expression (e.g., the UFBGEN label). The format for this operand is: UFB=filename. The UFB operand is placed after all READ modifiers (except the COND operand), and is separated from them by a comma.
COND	By specifying a value for COND you can make the execution of a READ function request conditional. The number or absolute expression that you specify becomes the first operand of the JSCI instruction by which the READ function is entered. The JSCI instruction is described in the VS Principles of Operation. DMS loads Register 1 with the UFB address even when the condition is not satisfied. The default value is 15.

7.4 THE WRITE FUNCTION REQUEST

The WRITE function request writes the record in the user record area (addressed by the UFB field UFBRECAREA) to the specified file. The record written can be either created in the user record area, or read into the user record area from another data file. You cannot use WRITE to update an existing record. You must use a REWRITE function request to write an existing record back to the data file from which it was read.

You can issue WRITE function requests in Output, Extend, I/O, and Shared modes. The WRITE function request is supported for all file types and access methods. You can issue a WRITE function request to files on disk or tape; you can not issue a WRITE function request on a workstation file (refer to Chapter 11 for details on workstation files).

Consecutive Files

Consecutive file creation and extension is performed using the WRITE function request. In Output mode, DMS writes records sequentially, beginning with the first record in the file. In Extend mode, DMS locates the end of the file data and begins sequentially writing records following the last record in the file.

If a consecutive file is open in I/O or Shared mode, when you issue a WRITE function request DMS automatically writes the record to the end of the file. DMS adds records to the end of the file in the sequence that the records are received. DMS issues a File Status '34' if you exceed the available file space. Multiple users can concurrently write records to a consecutive file opened for I/O update. Multiple users can also use WRITE function requests to output records to a consecutive log file open in Shared mode. Log files and file sharing are described in Chapter 8.

Relative Files

If you write records to a relative file in Output or Extend mode, DMS writes them into sequentially numbered record slots. If a relative file is in I/O mode, the WRITE function request writes records randomly, based on the Relative Record Number (RRN) stored in the field addressed by UFBKEYAREA. If the RRN refers to an existing record, DMS rejects the WRITE operation with a File Status '22'. A WRITE function request does not affect the file's current record pointer value.

If the RRN you specify is greater than the RRN of the last allocated record slot, DMS allocates as many additional extents as necessary (up to 13 extents) to be able to write the record in the appropriately numbered slot. When DMS allocates an additional relative file extent, it sequentially initializes each record slot from the beginning of the extent to the record slot location of the new record. If after allocating all 13 extents the relative record number you specified is still greater than the RRN of the last allocated record slot, DMS rejects the WRITE operation with a File Status '34'.

You can successfully perform a relative file WRITE operation without writing any data. If you issue a WRITE for a record with a RECSIZE of zero, DMS creates a zero-length record. Although a zero-length record contains no data, it does occupy a record slot and prevents subsequent writes to that slot.

You can write a record at the end of a relative file in I/O mode by issuing a WRITE function request with an EOF modifier. A WRITE EOF locates the last record slot containing data (including zero-length records), and writes the record in the next record slot. Following a

WRITE EOF, DMS returns to writing records randomly by relative record number. To write a series of records to the end of the file, you must specify a WRITE EOF for each record written. A WRITE EOF does not affect the currency of subsequent operations.

Indexed Files

You can sequentially write records into an indexed file in Output mode. Records written to an indexed file in Output mode must be in ascending order by primary key value. If you attempt to write an indexed record out of primary key sequence, DMS returns a File Status '21'.

If an indexed file is in I/O or Shared mode, the WRITE function request writes records randomly based on primary key value. DMS locates the primary key within the record, and uses its value to determine where to place the record in the file. Records written to an indexed file must contain a unique primary key value. If you attempt to write a record with a duplicate primary key value, DMS returns a File Status '22'.

When you write an alternate indexed record to a file, DMS also writes the 2-byte bit mask suffix for that record by copying the mask value placed in the AXD1MASK field. If the bits set in this mask are not a subset of the AXD1 PMASK, DMS rejects the WRITE function request with a File Status '98'. See Chapter 3 for details on the record mask and PMASK.

The WRITE function request clears the AXD1ALTINX field. Unless you reset this field, subsequent record access to an alternate indexed file is by primary key value.

In I/O and Shared modes the system detects all write errors at once. In Output mode, DMS does not detect write errors involving duplicate alternate index keys until the file is closed. See Chapter 14, DMS Error Routines, for details.

7.4.1 WRITE Function Request Syntax

The WRITE function request takes only one modifier, the optional EOF modifier for relative files. You must supply a UFB operand to each WRITE function request. You can also supply a COND operand that allows conditional writes.

```
[label]  WRITE   [EOF,]  UFB={(register)}   [,COND=number]
                            {expression}
```

The following is a list and explanation of WRITE function request modifiers and operands.

7.4.2 WRITE Function Request Modifiers

Modifier	Comments
EOF	You can only use the EOF modifier for relative files open in I/O mode. A WRITE EOF writes the record in the user record area into the record slot after the last data record in the file. Use an EOF modifier for every successive WRITE operation that writes a record to the end of the file. A WRITE EOF does not change the current record pointer value.

7.4.3 WRITE Function Request Operands

Operand	Comments
UFB	This operand is required for all WRITE statements. It specifies the address of a User File Block. You can supply it either as a register specification (where the register contains the UFB address), or as an expression (e.g., the UFBGEN label). The format for this operand is: UFB=address.
COND	By specifying a value for COND you can make the execution of a WRITE function request conditional. The number or absolute expression that you specify becomes the first operand of the JSCI instruction by which the WRITE function is entered. The JSCI instruction is described in the VS Principles of Operation. DMS loads Register 1 with the UFB address even when the condition is not satisfied. The default value is 15.

7.5 THE REWRITE FUNCTION REQUEST

You can use the REWRITE function request to update disk file records. DMS supports REWRITE processing in I/O and Shared modes. To update a record, you first perform a READ HOLD operation to read the record that you wish to update. Modify the data fields of the record in the user record area. You then issue a REWRITE function request to update the record. REWRITE obtains the record from the user record area (addressed by the UFBRECAREA field) and writes it to the file, overwriting the previous data and updating all indicators and pointers.

For consecutive file update, the file must be open in I/O or Shared mode. Variable length records cannot change in length during update processing. You cannot rewrite compressed records to a consecutive file.

You can use REWRITE to update records in a relative file. One method of updating a relative file record is to perform a READ HOLD operation in I/O mode, modify the data, then perform a REWRITE operation. Unlike consecutive files, the record rewritten to the relative file record slot does not have to be the same length as the record read in the READ HOLD operation, unless you originally defined the file as having fixed length records.

A second method to update records in a relative file is the REWRITE REL operation. You can only use REWRITE REL to update relative files. A REWRITE REL operation locates a relative file record by the relative record number that you supply in the field addressed by UFBKEYAREA, and overwrites the old record with the updated record. You do not have to precede a REWRITE REL with a READ HOLD operation.

When updating an indexed file, you must hold the record to be updated by issuing a READ HOLD operation in I/O or Shared mode, update the data in the user record area, and then perform a REWRITE operation. The REWRITE operation uses the primary key value to locate the record to be rewritten. For this reason you cannot modify the record's primary key value. You can rewrite fixed length, variable length, and compressed records in a indexed file. If you have changed the length of a variable length record, you must specify the new length in UFBRECSIZE before rewriting the record. You can change the length of an indexed variable length or compressed record within the limit established by the maximum record size.

DMS rewrites the alternate index bit mask suffix along with each alternate indexed record. If you wish to change the record's alternate key path assignment, your program must supply a new bit mask value to AXD1MASK before issuing the REWRITE. When you rewrite the record, the system modifies the alternate index trees to reflect changes to this indicator. You can rewrite a record to add it to or remove it from an existing alternate key path. However, if you set bits on the bit mask that are not set in the AXD1 PMASK, DMS rejects the REWRITE operation with a File Status '98'.

The REWRITE function request takes only one modifier, the REL modifier for relative files. You can supply UFB and COND operands to the REWRITE function request. The UFB operand is mandatory for all REWRITE statements; the COND operand allows conditional rewrites.

7.5.1 REWRITE Function Request Syntax:

The REWRITE function request syntax is as follows:

```
[label] REWRITE  [REL,]   UFB={(register)}   [,COND=number]
                              {expression}
```

7.5.2 REWRITE Function Request Modifiers

Modifier	Comments
REL	You can only use the REL modifier for relative files open in I/O mode. A REWRITE REL locates a record by relative record number (RRN) and overwrites that record with the contents of the user record area. To locate a record, you place the four-byte RRN in the area addressed by the UFB KEYAREA field, and then issue the REWRITE REL. If the RRN addresses an empty or nonexistent record slot, DMS rejects the REWRITE REL operation with a File Status '23'. You can update a variable length record with a record longer or shorter than the original record, as long as the record is not larger than the maximum record size. You can rewrite a record containing data with a zero-length record, and you can also rewrite a zero length record with a data record. A relative file REWRITE does not change the file's current record pointer value.

7.5.3 REWRITE Function Request Operands

Operand	Comments
UFB	This operand is required for all REWRITE statements. It specifies the address of a User File Block. You can supply it either as a register specification (where the register contains the UFB address), or as an expression (e.g., the UFBGEN label). The format for this operand is: UFB=address.
COND	By specifying a value for COND you can make the execution of a REWRITE function request conditional. The number or absolute expression that you specify becomes the first operand of the JSCI instruction by which the REWRITE function is entered. The JSCI instruction is described in the VS Principles of Operation. DMS loads Register 1 with the UFB address even when the condition is not satisfied. The default value is 15.

7.6 THE START FUNCTION REQUEST

The operation performed by the START function request depends on the access method and the modifier that you select. A START function request must always have a modifier. Depending on the modifier you request, a START can be used to:

- Change the processing mode for a file during program execution
- Position the DMS current record pointer
- Hold or release a record, group of records, or file in Shared mode. (Described in Chapter 8).
- Truncate a file
- Await completion of an I/O operation initiated while using the Physical Access Method (PAM).

The START operations available for RAM disk files, other than holding and releasing resources for Shared mode processing, are discussed in this section. The use of START for Shared mode processing is described in Chapter 8. The use of START in BAM and PAM is described in Chapter 10.

START function requests that change the access mode are only supported in Assembly language. Support of other START features varies among the different high-level languages. Refer to Chapter 5 for details of language support and syntax.

Table 7-2 summarizes the different modifiers that can be used with the START function request.

Table 7-2. Uses of the START Modifiers for Disk Files

	Currently in Input Mode	Currently in Output Mode	Currently in I/O Mode	Currently in Extend Mode
RAM Consecutive Files	BEGIN SKIP	EXTEND OUTPUT IO	BEGIN SKIP END EXTEND OUTPUT IO	EXTEND OUTPUT IO
RAM Relative Files	EQ GT GE LE LT	EXTEND OUTPUT IO	EQ GE GT LE LT EXTEND OUTPUT IO	EXTEND OUTPUT IO
RAM Indexed Files	EQ GT GE		EQ GT GE	
BAM	BEGIN	EXTEND OUTPUT IO	EXTEND OUTPUT IO	EXTEND OUTPUT IO
PAM	WAIT	WAIT EXTEND OUTPUT IO	WAIT	

The START WAIT is used only in PAM files, and is described in Chapter 10. START HOLD (and associated multiple modifiers elements) and START RELEASE are used for controlling resources in file sharing. They are described in Chapter 8.

7.6.1 START Function Request Syntax

```
[label] START {IO,}          UFB={(register)}     [,COND=number]
              {OUTPUT,}          {expression}
              {EXTEND,}
              {BEGIN,}
              {SKIP,}
              {EQ,}
              {GE,}
              {GT,}
              {LE,}
              {LT,}
              {END,}
             *{ HOLD,}
             *{ RELEASE,}
             *{ WAIT,}
```

** not described in this chapter*

7.6.2 START Function Request Modifiers

Modifiers	Comments
OUTPUT IO EXTEND	You can use these three START modifiers to change the access mode without closing and reopening a file. These modifiers can be used for relative files or consecutive disk files. These START function requests are not supported for consecutive log files opened in Shared mode, or consecutive files opened in I/O mode for shared processing. DMS permits you to open a file in any of these three modes then issue a START to specify another of these modes. Mode switching is only supported in Assembly language. START OUTPUT sets the DMS current record pointer to the beginning of the file. WRITE function requests issued after a START OUTPUT overwrite existing records at the beginning of the file. All existing records are logically deleted.

CAUTION

If you invoke a START OUTPUT, DMS reinitializes the file, scratching all existing data. Therefore you should use this function request with extreme caution. START OUTPUT cannot be used for updating files.

START EXTEND positions the UFB current record pointer to the end of the file. Subsequent WRITE operations sequentially add records to the end of the file. Only WRITE and START functions are permitted while you are in this mode. Issuing a START EXTEND while already in Extend mode has no effect.

START IO changes the mode from Output or Extend to I/O, permitting READ, REWRITE, and DELETE operations for updating the file. It sets a temporary end of file marker and sets the current record pointer to the first record in the file, facilitating sequential READ processing. At the conclusion of START IO processing, you can return to writing records to the end of the file by specifying START EXTEND. If you issue a START IO while already in I/O mode, DMS resets the current record pointer to the beginning of the file.

BEGIN
SKIP

Used only for consecutive files in Input mode, I/O mode, or Shared mode for file update. These modifiers specify the positioning of the DMS current record pointer, usually prior to the next READ operation.

NOTE

The START BEGIN and START SKIP function request can be used on both variable and fixed length records when used with Operating System Release 6.10 or later. On earlier operating system releases, START BEGIN and START SKIP can only be used on files containing variable length records.

START BEGIN positions the current record pointer to the first record in the file. START SKIP instructs DMS to skip the number of records specified in the word addressed by UFBKEYAREA.

If you specify a START SKIP with a value beyond the end of the file, DMS positions the current record pointer to the end of the file. If you specify a START SKIP with a value before the beginning of the file, DMS positions the current record pointer to the beginning of the file.

A READ issued after a START SKIP (with a signed binary number "n" specified for the KEYAREA) will:

1. skip over "n" records and read the record after them (n greater than 0)

2. merely read the next record (n = 0)

3. reread the current record (n = -1)

4. read a preceding record (n less than -1).

Example 7-6 demonstrates the use of the START SKIP function request.

Example 7-6. The START SKIP Function Request

```
OPEN   UFB=ZOOFILE
START  SKIP,UFB=ZOOFILE
READ   UFB=ZOOFILE
.
.
.
STATIC
ZOOFILE   UFBGEN FORG=CONSEC,VLEN=YES,KEYAREA=NUMSKIP
NUMSKIP  DC     F'+2'
```

EQ
GT
GE
LT
LE

You can use these START modifiers for positioning the current record pointer within the file. The modifiers listed are: Equal to, Greater Than, Greater than or Equal to, Less Than, and Less than or Equal to. START LT and START LE can only be used with relative files. The other modifiers can be used for both relative and indexed files.

You use these START modifiers with relative files to locate a record within the file by Relative Record Number (RRN). To perform one of these START requests, the relative file must be open in Input or I/O mode, and you must place the RRN in the area addressed by UFBKEYAREA before issuing the START request.

You use these START requests to locate records containing data in a relative file; empty record slots are ignored. If a START EQ equates to an empty record slot or a RRN that does not exist, DMS rejects the request with a File Status '23'. If you specify a START GT, GE, LT, or LE, DMS locates the first record that fulfills the condition specified in the START modifier. If no such record exists, DMS rejects the operation with a File Status '23'. Using one of these file position START function requests changes the file's current record pointer value.

You use these START modifiers with indexed files to locate a record by primary or alternate key value, especially when the exact value of the key sought is not known. The START function request positions a pointer to the record with a key value equal to or greater than the key value in the field addressed by KEYAREA. You cannot use the Less Than (LT) or Less than or Equal (LE) modifiers with indexed files.

You can use the START EQ, START GT, and START GE function requests for locating records by primary key value. Record location by primary key value is performed in the same way for primary indexed and alternate indexed files. Access by primary key value is the default option when an indexed file is opened.

You can also use these START function requests to locate records by alternate key value. Prior to issuing a START request, the program should modify the AXD1ALTINX field to specify the key path to be used for START access. Subsequent function requests access records along that alternate key path until an Open or Write operation resets the AXD1ALTINX field. If duplicate alternate key values are present on the key path selected, the START EQ locates the first record (record with lowest primary key) with that alternate key value.

If you issue a START EQ and DMS cannot find a record with the key value you specified, it returns a File Status '23'. If you issue a START GT or GE with a key location value greater than any existing key in the file, DMS returns a File Status '24'.

A READ (with no modifiers), or a READ HOLD can follow one of these START function requests to read the record indicated by the current record pointer. The example below locates and reads the first record with a primary key greater than '00100' in I/O mode. The current record pointer is positioned for sequential reads from that record.

Example 7-7. The START GT Function Request

```
CODE
OPEN     UFB=ZOOFILE,MODE=IO
MVC      PRIKEY,PRIVAL
START    GT,UFB=ZOOFILE
READ     HOLD,UFB=ZOOFILE
.
.
.
STATIC
ZOOFILE UFBGEN   FORG=INDEXED,RECAREA=ZOOREC,              X
                 KEYAREA=PRIKEY,KSIZE=5
ZOOREC  DS       0CL80
PRIKEY  DS       CL5
        DS       CL75
PRIVAL  DC       C'00100'
```

You can also use START GE to locate the first record in an indexed file. You accomplish this by specifying the primary key value as hexadecimal zeros.

If a comparison with the entire key field value is not desired, you can set UFBGKSIZE before issuing the START function request. The UFBGKSIZE (generic key size) specifies the number of characters to be considered in a comparison. After the START has been performed, UFBGKSIZE reverts to its default, which compares the entire key.

END

The START END truncates a data file at the location specified by the current record pointer value. You can only use this function request with consecutive files in I/O or Shared mode. Before issuing a START END to a consecutive file opened in Shared mode for update, you must hold the entire file for update. Unique situations may occur when performing a START END in Shared mode, refer to Chapter 15 for details.

CAUTION

START END deletes parts of data files or entire data files. Therefore you should use this function request with extreme caution. Programs using START END should check to insure that the current record pointer is on the proper record prior to issuing a START END.

You position the record location pointer with READ REL or START SKIP, then issue a START END to delete all records with higher relative record numbers than the current pointer location. A START END following a READ REL truncates all records following the record read. A START END following a START SKIP truncates all records following and including the record skipped to. A START END at the first record of the file (for example, following a START BEGIN) deletes all the records in the file creating a null file.

A START END eliminates all record data following the current record pointer; it does not release the space allocated for those records. All file extents remain allocated following a START END. A null file retains its User File Block and can be written to in Extend mode.

NOTE

START END is available with operating system Release 6.10 and all subsequent releases. Users with prior operating system releases cannot use START END.

7.6.3 START Function Request Operands

Operand	Comments
UFB	This operand is required for all START statements. It specifies the address of a User File Block. You can supply it either as a register specification (where the register contains the UFB address), or as an address expression (e.g., the UFBGEN label). The format for this operand is: UFB=address.
COND	By specifying a value for COND you can make the execution of a START function request conditional. The number or absolute expression that you specify becomes the first operand of the JSCI instruction by which the START function is entered. The JSCI instruction is described in the VS Principles of Operation. DMS loads Register 1 with the UFB address even when the condition is not satisfied. The default value is 15.

7.7 THE DELETE FUNCTION REQUEST

The DELETE function request deletes the last record read from a relative or indexed file on disk. The file you specify must be open for I/O or Shared mode processing. The last function request you issue before a DELETE must be a successful READ HOLD operation for an indexed file. You can delete records from a relative file without first performing a READ HOLD. You cannot delete records from a consecutive file.

You can delete records from a relative file using three methods. You can delete an individual record by issuing a READ HOLD for that record, and then issuing a DELETE function request with no modifier. You can also issue a DELETE REL to locate a record by relative record number and delete the located record. You can also delete multiple records by issuing a DELETE EOF, which deletes all records following the current record in the file.

You can only delete records containing data (including zero-length records) from relative files. A delete resets the record length indicator to zero, making the record slot available as an empty record slot. A relative file delete does not clear or zero-fill the record data area. You can delete zero-length records; you cannot delete empty record slots.

You delete a record from an indexed file by issuing a READ HOLD on the record, then issuing a DELETE with no modifier. When you delete a record, DMS shifts the remaining records within the block to close up space between records. DMS then resets the block length indicator to reflect the delete. DMS does not shift records between blocks following a DELETE. It makes a data block available as an empty block if you delete the last remaining record in the block.

If you delete the last record in an indexed file block, DMS updates the primary index block(s) to reflect this change. The alternate index paths specified for the deleted record are also modified as part of the DELETE function request.

A DELETE function request requires a UFB address operand to identify which file is being accessed. A COND operand allows conditional execution of the DELETE function request.

7.7.1 DELETE Function Request Syntax:

```
[label] DELETE   [REL,]  UFB={(register)}   [,COND={integer}              ]
                 [EOF,]      {expression}           {absolute expression}
```

7.7.2 DELETE Function Request Modifiers

Modifier	Comments
REL	You can only use DELETE REL with relative files open in I/O mode. A DELETE REL locates a record by relative record number (RRN) and deletes that record (resets its record length indicator to zero). To locate a record for deletion, you place the four-byte RRN in the area addressed by the UFB KEYAREA field, and then issue the DELETE REL. If the RRN addresses an empty or nonexistent record slot, DMS rejects the DELETE REL operation with a File Status '23'. You can delete zero-length records. A relative file DELETE does not change the file's currency pointer value.
EOF	The DELETE EOF truncates a data file at the current location pointer. You can only use this function request in I/O mode with relative files. You position the record location pointer with READ REL or START, then issue a DELETE EOF to delete all records with higher relative record numbers than the current pointer location. A DELETE EOF following a START deletes all records that follow the record located by the START. A DELETE EOF following a READ REL deletes all records following the record after the record read. A DELETE EOF at the beginning of the file does not delete the record in the first slot of the file. To delete all records in a file, use START OUTPUT. A DELETE EOF resets the file's E-Block and EREC indicators so that DMS treats deleted records as beyond the last data record in the file. If you then issue a WRITE function request for a record slot beyond EREC, DMS sets to zero all of the record length indicators between EREC and the record written. A DELETE EOF does not zero-fill deleted records or deallocate file extents.

7.7.3 DELETE Function Request Operands

Operand	Comments
UFB	This operand is required for all DELETE statements. It specifies the address of a User File Block. You can supply it either as a register specification (where the register contains the UFB address), or as an address expression (e.g., the UFBGEN label). The format for this operand is: UFB=address.

COND

By specifying a value for COND you can make the execution of a DELETE function request conditional. The number or absolute expression that you specify becomes the first operand of the JSCI instruction by which the DELETE function is entered. The JSCI instruction is described in the VS Principles of Operation. DMS loads Register 1 with the UFB address even when the condition is not satisfied. The default value is 15.

CHAPTER 8
SHARING DATA FILES

8.1 INTRODUCTION

DMS enables file sharing of consecutive and indexed disk files accessed using the Record Access Method (RAM). File sharing enables multiple users to perform concurrent updates to different records in the same file. The system controls file sharing to automatically maintain file consistency and to prevent deadlock situations.

DMS monitors and controls file sharing on a task basis. Each user can run one interactive task at a time and submit one or more task for background execution. A task can consist of one or more programs and linked subroutines.

A task initiates shared update processing by opening a file in Shared mode. Other tasks can also open this file in Shared mode. A task can claim exclusive (non-shared) update access to a file by opening a file in I/O mode. A file opened by a task in I/O mode cannot be opened by other tasks.

You can open a consecutive file in Shared mode in two ways. You open an existing consecutive disk file as a shared consecutive file by setting both the Shared mode (UFBF2SHARED) and the I/O mode (UFBF2IO) bits in the file's UFB. A consecutive file thus opened in I/O mode as a shared file can be concurrently updated by multiple users. Note that simply opening an existing consecutive file in Shared mode does not provide this I/O support; you must specify both I/O mode and file sharing. Shared I/O processing of consecutive files is supported in Operating System Release 6.20 and subsequent releases.

Multiple users can open a new consecutive disk file in Shared mode and sequentially write output records to the file. When you open a new consecutive file in Shared mode, DMS defines that file as a log file. When you open a log file in Shared mode, the Sharer opens the file in Output mode (if it is a new file) or Extend mode (if the file is an existing file that was created as a log file). This permits multiple users to write records to the end of the log file in chronological order. You cannot read or update records in a log file while it is open as a log file in Shared mode. A log file can be closed and then reopened as a shared consecutive file in I/O mode. Log files are described separately in Section 8.9 of this chapter.

You can open an indexed file for shared update processing by specifying MODE=SHARED. A shared indexed file has all of the functions of an indexed file opened in I/O mode, along with the additional Shared mode functions.

File sharing is not available in Block Access Method (BAM) or Physical Access Method (PAM). You cannot share relative files, files on NOVTOC diskettes, or files on storage media other than disk.

DMS file sharing allows each task to hold one data resource at a time. A resource can be a record, a file, or a group of records or files, but you must hold the entire resource as a single operation. Under normal DMS sharing, you cannot incrementally claim new resources during a program while continuing to hold previously claimed ones.

Each task can hold or free a resource at any point during file processing. For example, you can hold a record for update as part of the Read operation, and free that record as part of the Rewrite operation. This claim-as-needed sharing feature minimizes the impact of a record update on other tasks by minimizing the period of time during which the resource is unavailable to other tasks.

Under normal DMS sharing, a task holds one resource exclusively. This hold prevents other tasks from updating that resource (deleting or modifying a record) until the first task frees the resource. A task can add records to a shared file if the records are not being added to a resource held by another task.

To incrementally claim multiple records for update, you can use DMS/TX file sharing. When a task accesses a DMS/TX file in Shared mode it automatically invokes DMS/TX file sharing. The holding of resources in DMS/TX is identical to DMS record holding. However, DMS/TX does not require you to free a held resource before claiming another resource. DMS/TX maintains holds on multiple records for the duration of the task's transaction, rather than releasing each held record as it is updated. DMS/TX features are described in the VS DMS/TX Reference.

Ordinary DMS permits you to hold only one record or group of records at a time. To hold more than one record, you must claim all members of the group of records at the same time. However, some existing user programs use the extension rights feature of DMS to incrementally claim multiple resources on an as-needed basis. This extension rights feature is described in Chapter 15 of this manual. DMS/TX sharing provides multiple record sharing superior to the use of extension rights. Use DMS/TX, rather than extension rights for the coding of new applications that incrementally claim multiple resources.

8.2 RESOURCE HOLDING OVERVIEW

This chapter describes the three types of DMS data resource holds you can use DMS (without extension rights) to perform:

- Implicitly holding a single record, using a Read Hold statement.

- Explicitly holding a generic key group of records or a single record by primary key value.

- Explicitly holding an entire file.

In addition, DMS provides three optional features that you can use when holding resources:

- The list option, which allows a task to simultaneously hold several resources. Each task can simultaneously hold a list of resources in one or more files.

- The hold for retrieval option, that allows multiple tasks to hold the same record for read access, but allows no task to change or delete the held record.

- The timeout exit option that allows the program to continue processing if a desired resource is unobtainable.

The Timeout option is available to both implicitly and explicitly held resources. The Hold List and Hold for Retrieval options are only available to explicit resource holds.

All resource holds are supported for indexed files. Unless otherwise noted, DMS supports these resource holds for shared consecutive files as well.

8.2.1 File Sharing Terms and Concepts

A task holds and frees resources within a shared file. A resource is either a single record, a group of logically contiguous records within a file that are related by their generic key value, or the entire file.

DMS provides two types of record holds: implicit and explicit. The system automatically applies an implicit hold when you invoke a Read Hold. An implicit hold is automatically released if you attempt to implicitly hold another record by issuing a Read Hold, or if you update or delete the record held.

An explicit hold consists of a statement that explicitly holds a resource and a second statement that explicitly frees the held resource. An explicit hold pre-claims a resource by naming it explicitly, prior to any statement to read or modify the data values of the held resource.

All holds are issued to the Sharer. The Sharer runs as a dedicated system task in background, with its own Segment 2 space allocation. Because all tasks request holds of the Sharer, it is able to prevent tasks from holding the same resource. The Sharer keeps track of which tasks are holding which resources in its Segment 2 area. Applying a hold does not read or modify the data file itself. Therefore, a hold issued for a non-existent resource is a legitimate hold, and must be released before you can apply another hold.

8.3 IMPLICIT HOLDS -- THE READ HOLD OPERATION

You can hold individual records implicitly. During an implicit record hold, you issue the hold as part of the Read statement, rather than as a separate statement. In this way you can claim records as needed; you do not hold the record until you actually need it. The Read statement may locate the record to be held sequentially, by Relative Record Number (consecutive files), or by primary or alternate key value (indexed files). Files and generic key groups can only be held explicitly; you cannot hold them implicitly.

The task does not have to issue an instruction to release an implicit record hold. The system releases a implicitly held record when any of the following occurs:

- The task successfully rewrites the held record.
- The task successfully deletes the held record.
- The task initiates a write operation on any shared file.
- The task invokes a Read Hold operation for another record in any file.
- The task invokes an explicit hold on any resource.
- The task issues a release command for an explicit hold.
- The file which contains the held resource is closed.

DMS supports implicit record holds in all VS languages that support the Shared mode. In Assembly language, you can code an implicit record hold by issuing a READ HOLD instruction. You can combine the HOLD modifier with other READ modifiers in parentheses as elements of a multiple modifier. See Chapter 6, section 3 for details.

8.4 EXPLICIT HOLDS

You can use an explicit hold operation to hold an entire file or a generic key group. You can hold a single record as a special case of holding by generic key. Even if a record is explicitly held, you must also implicitly hold it if it is to be rewritten or deleted. An explicit hold must be explicitly released by issuing a Release statement specifying the file in which a resource is held.

The system releases a explicitly held resource when either of the following occurs:

- The task issues a Release command for the file.
- The file which contains the held resource is closed.

8.4.1 Holding a Shared File

A task can open a file in Shared mode, and then issue a hold for the entire file. Holding a file in Shared mode provides exclusive update rights to all of the records in the file (unless you specify hold for retrieval). The advantage of holding a file in Shared mode, rather than simply opening it in I/O mode, is that in Shared mode you can release the file to other users without the overhead of closing the file. Other users can open the file in Shared mode while you are holding it, but they must wait for you to release the file before they can claim any resources in that file. You can release an explicitly held file by issuing a release statement for the file or a general release statement that releases all resources, depending on which language you are using.

An explicit hold for shared files is available in COBOL and Assembly language. File holding is supported in RPG II for consecutive files only. In COBOL, you can issue a file hold for indexed files by coding a HOLD statement (Format 2) without the INITIAL clause. You release a held file by issuing a FREE ALL statement.

In Assembly language, you invoke a hold on a Shared file by issuing a START HOLD command, as shown in Example 8-1.

Example 8-1. Holding and Releasing a File

```
START  HOLD,UFB=ZOOFILE
.
.
.
START  RELEASE,UFB=ZOOFILE
```

A held file is released by issuing a START RELEASE command for that file. The UFB address of the file can be specified as either an address or a register. Holding and releasing a file does not change the file's current record pointer value.

8.4.2 Holding Multiple Records by Generic Key

You can explicitly hold a logically consecutive group of records that share a common range of values. In consecutive files, these values are Relative Record Numbers; in indexed files, these values are primary key values.

Consecutive Files

To hold a range of records in a consecutive file, you specify a Relative Record Number (RRN), and then issue a hold on all records in the file with a RRN greater than or equal to the one you specified. DMS performs an explicit hold on this range of records that prevents other tasks from updating or deleting any of the held records.

When you hold a range of records, all actual or potential records with RRNs higher than the one specified are held. Other tasks cannot add new records to the end of the file. You can hold a range of records beginning with a RRN greater than any existing RRN. This prevents other tasks from extending the file to include records with that range of Relative Record Numbers.

An explicitly held range of records must be explicitly released. You must issue a general release statement to release all resources held by your task in a particular file.

Holding a range of consecutive file records is supported in RPG II and Assembly language. In RPG II you hold a range of records using the HOLD statement, supplying the RRN to the factor1 field.

In Assembly language, you hold a range of consecutive file records by first supplying a four-byte RRN to the area addressed by UFBKEYAREA. Then you issue a START HOLD,RANGE. A START HOLD,RANGE performs the actual holding of the records with Relative Record Numbers greater than or equal to the one specified.

To release a range of consecutive file records in Assembly language use a START RELEASE statement. START RELEASE simultaneously releases all records held by your task within the specified file. The only required operand for the START RELEASE is the UFB address of the file. The UFB can be specified as an address or a register.

Indexed Files

You can hold a range of records in an indexed file by generic key value. Records related by generic key all have the same value for the initial character or characters of their primary key fields. For example, if a file uses employees' names as a primary key field, all last names that begin with "Mc" share a common generic key. A generic key can be the full length of the primary key; such a generic key would hold a single record.

It is possible to hold a generic key group that contains no records. One use of this hold would be to prevent other tasks from writing new records within a particular generic key.

An explicitly held generic key group must be explicitly released. You issue a general release statement to release all resources held by your task in a particular file.

VS COBOL, RPG II, and Assembly language support the holding of records by generic key. In COBOL you can hold a generic key group by specifying the key field in the INITIAL and CHARACTERS OF phrases of the HOLD statement (Format 2). You release a generic key group by issuing a FREE ALL statement. In RPG II you issue a HOLD statement and supply the generic key value to the factor1 field to hold records by generic key.

In Assembly language, you hold a generic key group by performing three operations. First you establish the value of UFBGKSIZE, which determines how many characters of the primary key will comprise the generic key. Then you supply a partial or complete primary key value to establish a target key for the generic key group. Finally, you issue a START HOLD,RANGE. A START HOLD,RANGE performs the actual holding of the generic key group.

For example, in a file of employees that uses last name and employee number as a primary key, you can hold the generic group of records of people whose names begin with "Mc" as shown in Example 8-2.

Example 8-2. Holding and Releasing a Generic Key Group

```
MVI      UFBGKSIZE,X'02'
MVC      PRIKEY,=C'McDuffy12345'
START    HOLD,RANGE,UFB=EMPLOY
   .
   .
   .
START    RELEASE,UFB=EMPLOY
```

In Example 8-2, you first establish the generic key search field as two characters wide in the User File Block field UFBGKSIZE for the specified file. (The method used to modify UFBGKSIZE depends on how you established UFB addressing, as described in Chapter 6.) You then specify that these two characters will be 'Mc' by providing a valid primary key value (a full key value or a left-justified partial key value) that begins with those characters in the field addressed by UFBKEYAREA. Finally, you issue the command to hold the generic key foı the specified file.

If you specify a generic key size of zero, or a generic key size equal to or larger than the length of the key field, DMS interprets the generic key size as the full key length. It holds the record with that primary key value, holds that primary key value for a non-existent record and prevents other users from writing a record with that key value.

To release a generic key in Assembly language, you can use a START RELEASE statement. This START RELEASE simultaneously releases all generic key groups held by the task within the specified file. The only required operand for the START RELEASE is the UFB address of the file containing the generic key.

8.4.3 Holding A Single Record

To explicitly hold a single record, you must specify a Relative Record Number (RRN) or a primary key value that uniquely identifies a single record.

In consecutive files, you first supply the desired record's four-byte RRN to the area addressed by UFBKEYAREA. You then issue a START HOLD,EQUAL function request. You can hold an RRN even if no record exists with that RRN. Holding a non-existent record prevents other tasks from creating a record with that RRN; thus you can establish a temporary maximum file size. A START HOLD,EQUAL is normally a Hold for Update; you can specify RETRIEVAL as an additional modifier element to make it a Hold for Retrieval. Issuing a START HOLD,EQUAL does not change the value of the file's current record pointer. START HOLD,EQUAL is an explicit hold; you must explicitly release the record held by issuing a START RELEASE.

To explicitly hold a single record in an indexed file, specify a generic key size equal to the full size of the primary key field, then issue a hold by generic key and supply a primary key value to the area addressed by the UFBKEYAREA field. This operation holds a generic key that contains a single record identified by its primary key value. You can also hold a non-existent record in the fashion. This prevents another task from writing a new record with that primary key value. You must explicitly release an explicitly held generic key group.

8.5 INTERACTION BETWEEN FILE HOLDS

In most cases, you can hold only one DMS resource at a time. Generally, in order to hold a resource you must first release any previously held resource. However, this rule has several exceptions. If the second resource you wish to hold is a non-disjunctive subset of the first resource (i.e., it is completely contained within the first held resource) both holds are allowed. For example, if you are holding a generic key group, you can issue a hold request for one of the records within that generic key group without releasing the generic key hold. The possible combinations of non-disjunctive holds are shown in Table 8-1.

Table 8-1. Hierarchy of Resources
Non-disjunctive Holds

Previously Held Resource	Current Resource Hold Issued			
	Hold File	Hold Generic Key	Hold Record READ HOLD	Hold Record START HOLD
Hold File	maintain previous hold	maintain previous hold	maintain previous hold	maintain previous hold
Hold Generic Key	ERROR	release previous hold *	maintain previous hold	maintain previous hold
Hold Record READ HOLD	release previous hold	release previous hold	release previous hold	release previous hold
Hold Record START HOLD	ERROR	ERROR	ERROR	maintain previous hold

* for subsets only.

The Hold Record START HOLD shown in Tables 8-1 and 8-2 is the special case of the hold by generic key in which the entire primary key is supplied as the generic key.

As shown in Table 8-1, the Sharer automatically releases all implicit record holds when you issue another hold request, either explicit or implicit. The Sharer issues a File Status '86' if you request an explicitly held resource higher in the hierarchy (a superset); the Sharer maintains explicit holds if you request a resource lower in the hierarchy (a subset).

An explicit hold cannot be released by issuing an implicit hold. You must release an explicit hold by issuing either another explicit hold or an explicit release.

A hold on a generic key group is an explicit hold. If a hold on a generic key group is followed by another generic key group hold, the situation falls into one of the following four categories:

- The first and second generic key groups are identical. In this case the Sharer releases and immediately reapplies the hold.

- The second generic key group is a subset of the first generic key group. Since the task already holds all records in the second generic key group, the Sharer maintains the first hold and ignores the second hold.

- The second generic key group is a superset of the first generic key group (e.g., names beginning with M are a superset of names beginning with Mc.) This type of hold is an error. You must release the first explicit generic key hold before issuing a superset hold.

- The first and second generic key are disjunctive, holding different sets of records. You must release an explicit hold before any disjunctive hold (explicit or implicit) can be applied. You can, however, simultaneously hold several disjunctive generic key groups by specifying them is a list, as described in Section 8.6.

If there are no records common to two hold operations, the operations are disjunctive. The results of issuing two disjunctive record holds are shown in Table 8-2.

Table 8-2. Disjunctive Holds

Previously Held Resource	Current Resource Hold Issued			
	Hold File	Hold Generic Key	Hold Record READ HOLD	Hold Record START HOLD
Hold File	ERROR	ERROR	ERROR	ERROR
Hold Generic Key	ERROR	ERROR	ERROR	ERROR
Hold Record READ HOLD	release previous hold	release previous hold	release previous hold	release previous hold
Hold Record START HOLD	ERROR	ERROR	ERROR	ERROR

8.6 HOLDING A LIST OF RESOURCES

You can hold multiple shared consecutive or indexed files and/or groups of records by requesting these holds as part of a Hold List operation. A Hold List operation simultaneously holds the files and generic key groups that are specified as items on a list. You can separately release these listed resources on an individual file basis.

You can only hold explicitly held resources as members of a list. You initiate a list by specifying a List option on the explicit hold statement for a resource. Specifying the List option sets a User File Block flag that indicates a hold list operation in progress. Additional resources are added to the list by issuing an explicit hold with the List option for each resource. The Sharer does not hold any of the listed resources until the list is completed. You complete a list by explicitly holding a resource without specifying the List option. When the list is completed, the Sharer simultaneously holds all of the resources listed (including the one without the list option) and it resets UFBVLIST to zero.

Since all items in a list are held simultaneously, the sequence of items in a list is unimportant. DMS automatically handles duplicate items and non-disjoint subsets. Mixing Hold for Update and Hold for Retrieval items as non-disjoint sets results in an error (these features are further described in Section 8.7). All items in a list must be held at the same time. If the Sharer cannot hold all items in the list, it holds none of the items in the list.

Statements other than explicit holds are invalid if you issue them while an incomplete list is pending. Issuing an explicit release statement releases all held items for the specified file and deletes a pending list.

You can use the timeout option when holding a list of resources. You specify the timeout when holding the last item on the list. If the Sharer cannot hold all items on the list within the number of seconds specified as the timeout, the list is not held and DMS takes the timeout exit specified for the last listed item.

Under DMS/TX, the items in a list are individually held as they are specified, rather than simultaneously. The use of the List option is not recommended under DMS/TX. Refer to the VS DMS/TX Reference for details.

The List option is supported in COBOL, RPG II, and Assembly language. In COBOL it is provided as an option of the HOLD statement (Format 2). You release a held list using a FREE ALL statement. In RPG II you perform a hold list by using the HOLDL instruction. The FREE instruction releases all held resources.

In Assembly language the List option is included as part of the Start Hold statement, as shown in Example 8-3.

Example 8-3. Holding and Releasing a List of Three Items

```
START   HOLD,LIST,UFB=ONEFILE
START   HOLD,(RANGE,LIST),UFB=TWOFILE
START   HOLD,UFB=THREEFILE
.
.
.
START   RELEASE,UFB=ONEFILE
START   RELEASE,UFB=THREEFILE
.
.
.
START   RELEASE,UFB=TWOFILE
```

In Example 8-3, the first START HOLD initiates the list operation, specifying a file to be held. The second START HOLD continues the list by specifying a hold on a generic key group in a second file. A multiple modifier in parenthesis is used for generic key holds in a list. The third START HOLD does not contain the LIST option. DMS reads this as the final item on the list. Upon reading this instruction, DMS performs the actual holds, simultaneously holding the ONEFILE and THREEFILE files, and the generic key group in file TWOFILE. A START RELEASE statement releases the resources held by the task for a particular file. You can release items held as a list individually on a per-file basis.

8.7 THE HOLD FOR RETRIEVAL OPTION

A normal DMS hold is a Hold for Update. This means that when a task holds a resource, no other task may hold, rewrite, or delete any record within that held resource.

DMS also provides a Hold for Retrieval option. If you explicitly hold a resource with a hold for retrieval, other tasks can also perform Read Holds on the records held by your task. However, neither you nor the other tasks can rewrite or delete the records that you are holding for retrieval. For the duration of the hold, all tasks can only use the resource for data retrieval.

Hold for Retrieval allows you to prevent modification of file records and maintains data consistency without restricting access to the file by other tasks. When you specify a resource as held for retrieval, Read Hold operations for other tasks on that resource are processed as ordinary Read operations.

Explicit Holds for Update and Holds for Retrieval are incompatible when performing non-disjunctive holds in DMS. For example, an error occurs if you hold a file for retrieval, then attempt to hold for update a generic key group in that file. You can perform disjunctive Holds for Retrieval and Holds for Update on the same file. The explicit release statement releases both retrieval and update holds.

Hold for Retrieval is supported in COBOL, RPG II, and Assembly language. In COBOL it is an option of the Hold statement (format 2). In RPG II you must follow the filename specified in every HOLD instruction by a U or R, indicating respectively Hold for Update or Hold for Retrieval.

In Assembly language a hold for retrieval is performed using the START instruction with a multiple modifier.

Example 8-4. Holding a File for Retrieval

```
START  HOLD,RETRIEVAL,UFB=ZOOFILE
.
.
.
START  RELEASE,UFB=ZOOFILE
```

A START HOLD,RETRIEVAL holds an entire file. You can also issue a START HOLD,(RANGE,RETRIEVAL) to hold a range of consecutive file records or an indexed file generic key group. You can also issue a START HOLD,(EQUAL,RETRIEVAL) to hold an individual consecutive file record, or a START HOLD,(LIST,RETRIEVAL) to hold a list of resources for retrieval.

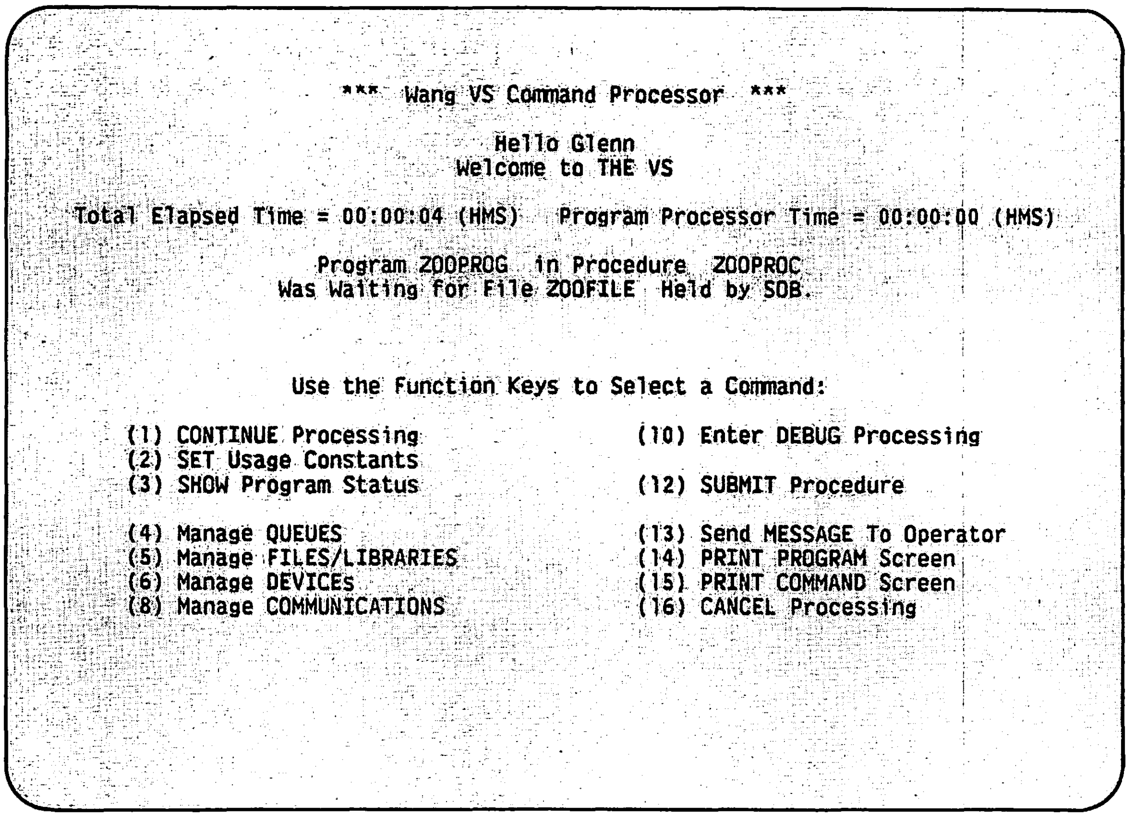
8.8 THE TIMEOUT OPTION

DMS provides a timeout option for both consecutive and indexed files. When you issue a hold, DMS attempts to apply the hold within the time specified in the timeout option.

8.8.1 Task Waiting Without the Timeout Option

An attempt to hold a resource may not be successful because another task is holding the desired resource, or a subset of it. In this situation, the DMS default is to place the task in a wait state while awaiting the release of the held resource. DMS suspends a list operation until all items on the list are simultaneously free. A task remains in a wait state until it can hold the specified resource(s) or it is cancelled by the operator.

You can determine what resource an interactive task is waiting for by pressing the HELP key. Pressing the HELP key suspends processing and returns you to the Command Processor. If the task was awaiting a shared resource when it was suspended, the Command Processor screen displays a message, as shown in Figure 8-1.



```
                    ***  Wang VS Command Processor  ***

                              Hello Glenn
                            Welcome to THE VS

   Total Elapsed Time = 00:00:04 (HMS)   Program Processor Time = 00:00:00 (HMS)

                    Program ZOOPROG  in Procedure  ZOOPROC
                  Was Waiting for File ZOOFILE  Held by SOB.


                   Use the Function Keys to Select a Command:

      (1) CONTINUE Processing                 (10) Enter DEBUG Processing
      (2) SET Usage Constants
      (3) SHOW Program Status                 (12) SUBMIT Procedure

      (4) Manage QUEUES                       (13) Send MESSAGE To Operator
      (5) Manage FILES/LIBRARIES              (14) PRINT PROGRAM Screen
      (6) Manage DEVICES                      (15) PRINT COMMAND Screen
      (8) Manage COMMUNICATIONS               (16) CANCEL Processing
```

Figure 8-1. HELP Processor Screen

8.8.2 Task Waiting Using the Timeout Option

DMS provides an optional timeout feature to avoid indefinite waits for resources held by other tasks. The timeout feature provides a timeout duration, and a timeout exit in each file's User File Block.

The timeout duration is stored in the User File Block field UFBTIME. You can set the duration field to any value from 0 to 255 seconds. If you specify a value of zero, DMS will wait indefinitely. Once set, the timeout value remains in effect for resources in that file until the file is closed or UFBTIME is reset.

DMS automatically applies the specified timeout value to the following operations:

- Read Hold (implicit record holds)
- Hold File or Hold Generic Key (explicit holds)
- Write operations
- Holding extension rights (refer to Chapter 15)

DMS/TX processing supports timeout processing for rewrite and delete operations as well. Refer to the VS DMS/TX Reference for details.

When you invoke one of the above operations, DMS checks the timeout exit field of the UFB. This UFBTIMEEXIT field has a default value of zero. If DMS finds a zero value in UFBTIMEEXIT, no timeout support is provided, and a task waiting for a resource will wait indefinitely.

If you supply a non-zero value to the UFB timeout exit field, a timeout exit is taken. DMS inspects the UFBTIME field for the number of seconds to maintain an item on the wait queue before taking the timeout exit. If the UFBTIMEEXIT value is non-zero, and if UFBTIME is set to zero seconds, DMS immediately takes the timeout exit if it cannot immediately hold the requested resource. If the UFBTIMEEXIT value is non-zero and the UFBTIME value is non-zero, DMS waits the hold request on a queue for the number of seconds specified in UFBTIME, then takes the timeout exit.

A timeout exit is a user-supplied program address that you specify in the User File Block's UFBTIMEEXIT field. If you provide a timeout exit, program execution branches to the address specified in the timeout exit and continues. DMS sets a File Status value of '70' to indicate that a timeout exit was taken.

When a list of resources is held, you specify the timeout value with the last item on the list. The system applies this timeout value to all of the resources on the list. If it cannot hold the complete list within the specified number of seconds, no holds are applied and the timeout exit is taken. The use of the timeout for Hold List processing of DMS/TX files is somewhat different; refer to the VS DMS/TX Reference for details.

File positional currency is not reliable after a timeout exit has been taken. You must reposition the file pointer before performing sequential operations.

After a timeout exit, you can access the log-on ID of the current holder of the resource that was unavailable. The name of the resource requested, and the user ID of the current holder of that resource can be extracted from the UFB. The current user's ID is located in the User File Block field UFBHOLDID. This field is maintained by the system, and should never be written to by the user.

Timeout processing is supported in COBOL, RPG II, and Assembly language. COBOL provides the timeout option with the HOLD, READ, and WRITE statements for indexed files.

In RPG II the number of seconds to wait before a timeout is specified by the global operator *SECS. If you have established a timeout, the timeout exit is provided via the Resulting Indicator status field of the HOLD statement.

In Assembly language, you establish the timeout duration and timeout exit options by supplying values directly to the UFBTIME and UFBTIMEEXIT fields. The same timeout values will be used by all of the file's operations that support timeout until the file is closed or you modify the UFB values.

Example 8-5. Establishing Timeout Values

```
MVI    UFBTIME, X'06'
LA     R6,TIMEROUT
ST     R6,UFBTIMEEXIT
READ   HOLD,UFB=ZOOFILE
```

The method you use for modifying the UFBTIME and UFBTIMEEXIT fields depends upon the method you used to establish the address of the UFB, as described in Chapter 6.

8.9 LOG FILES

A log file is a consecutive data file on disk that functions as an Output mode file that can be concurrently written to by multiple tasks. Log files are opened in Shared mode by multiple tasks; a task cannot open a log file in any other mode until all tasks close the log file. DMS adds records to the end of a log file in chronological order. The Write operation is the only operation that can be performed on a log file while it is open in as a shared log file; a task cannot read a log file while it is open as a shared log file.

A log file is a special type of consecutive file, containing variable length records. The record format for log files is the same as for normal consecutive files. The block structure of a log file differs slightly from the block structure of a normal consecutive file. You can process an existing log file as an ordinary consecutive file in Input, Extend, and I/O modes. You can also open an existing log file as a shared consecutive file for I/O processing. A log file cannot be simultaneously open as both a shared log file (output processing) and a shared consecutive file (update processing).

You create a log file by opening a new consecutive file in Shared mode, rather than Output mode. Writing the initial records to the file in Shared mode flags the file as a log file in the UFB. To add records to a log file you open an existing log file in Shared mode. DMS automatically locates the end of the file and writes new records to extend the file.

Log file support is provided for all VS languages that support the Shared mode. In Assembly language, you can create a log file as part of the OPEN statement, as shown in Example 8-6.

Example 8-6. Creating a Log File in Shared Mode

```
                 OPEN UFB=NEWZOO,MODE=SHARED
                 .
                 .
                 .
                 STATIC
        NEWZOO   UFBGEN  FILENAME=@ZOOFILE,FORG=CONSEC,VLEN=YES
```

You can also create a log file for your exclusive use by setting the UFBFLAGSLOG bit, and then opening a new consecutive file in Output mode, as shown in Example 8-7.

Example 8-7. Creating a Log File in Output Mode

```
             CODE
             OI   UFBFLAGS,UFBFLAGSLOG
             OPEN UFB=ZOOFILE,MODE=OUTPUT
```

This provides you with all of the features of ordinary log file processing except file sharing. You can, however, close the file and reopen it in Shared mode to enable multiple users to extend the file.

A log file created in Output mode by setting the UFBFLAGSLOG bit can contain compressed or non-compressed records. A log file created by opening a consecutive file in Shared mode cannot contain compressed records.

DMS provides two special features to prevent loss of log file data due to a system crash. These two features are log file recovery and record write-through.

When a system crashes, the system may not have updated the VTOC of an open file to reflect changes made to the file data. Log file recovery allows you to successfully reopen a log file following a system crash. DMS performs log file recovery automatically when an Open in any mode is attempted on a crashed log file. DMS inspects log file flags in the data file to determine the end-of-file location. DMS recovers both the record count and the EOF indicator; no message is issued.

Record write-through writes individual records to the disk file directly, rather than accumulating records in a buffer. This prevents the loss of the most recently written record(s) when a system crash occurs. Performing a disk I/O operation for each record written enhances security against loss of records, but deprives you of the performance advantages of I/O buffering. Record write-through is optional; you can create a log file with or without the write-through feature. You can invoke write-through by specifying an @ as the first character of the log file's name. Write-through is a feature of the log file that is in effect for all write operations in Output, Extend, or Shared mode.

After opening a log file, you use standard DMS WRITE and CLOSE statements to add records to the file.

CHAPTER 9
DMS EFFICIENCY CONSIDERATIONS

9.1 PERFORMANCE IMPROVEMENT METHODS

DMS provides two methods for improving performance of data files. The first method, packing density, should be employed when you plan to add a substantial number of records to an indexed or alternate indexed file after creating the file. You can use the second method, buffering, to reduce I/O overhead on any file. Two buffering strategies are available: the large buffer strategy for consecutive or relative files, and the buffer pooling strategy for keyed access to an indexed file.

Other considerations that have an effect on performance are described elsewhere in this manual. The selection of appropriate record and file types is described in Chapters 2 and 3. The desirability of minimizing the number of alternate keys and the lengths of all keys is explained in Chapters 3 and 15. Running COPY and other utilities to reorganize a file is described where applicable.

9.2 PACKING DENSITY

The packing density is the percentage of a block initially used for writing records. When you set a packing density of less than 100%, DMS leaves some space within each block for future records added to that block.

9.2.1 User Interface

When creating or copying data files, you must set two packing density fields, IPACK and DPACK. IPACK is the packing density for primary index tree blocks; DPACK is the packing density for data blocks.

The IPACK and DPACK field are options for UFBGEN (used to create the file), and three file copying utilities: COPY, TAPECOPY, and IBMCOPY (IBM-format diskettes). The COPY utility uses Record Access Method (RAM) to reblock records for a different packing density. This results in a slower copy operation than an unmodified COPY, which executes in Block Access Method (BAM).

9.2.2 The Default (100%) Option

The packing density defaults are 100% in all cases. A packing density of 100% packs as many records or index table entries as possible into a 2K block. (Do not confuse packing density with packed decimal format; the packing density does not change the data representation from standard ASCII bytes). While a 100% packing density provides the most compact file (fewest blocks allocated), and thus the smallest index trees and fastest access, updates to a 100% file tend to be costly and inefficient.

For example, to add one record to the middle of a file with 100% packing density for data and index blocks would require DMS to perform the following steps:

1. Locate the block to be updated by primary key, determine that there is insufficient room to put the record in the data block.

2. Perform a block split on the data block, moving one-half of the data to an available block at the end of the file. If no blocks are available, DMS must allocate a new extent.

3. Change the block addressing for both blocks created by the block split in the low-order index block. Since the index block is packed 100%, there is no room in the index block for the additional data block table entry. DMS must also split the index block.

4. Split the low-order index block, placing half of it in an available block at the end of the file. Change the index block addressing for the next higher index block level. Because the packing density is 100%, this requires a block split on this level as well, which necessitates readdressing and block splitting successively on every higher level of the index tree.

5. The same process is repeated for each alternate key path.

Obviously, you should not use a 100% packing density for any indexed or alternate indexed file that you expect to update by adding numerous records. The initial rapid access time for a 100% file degrades rapidly as you add more records to the block, causing block splits to distant locations on the disk. You should establish an indexed file with a packing density of 100% if you plan to update it only by replacing one fixed length record with another, or by adding records with primary key values greater than the records already in the file.

9.2.3 The DPACK Field

The DPACK, or data packing field, determines how much of a data block DMS should initially allocate for writing data, and how much it should retain for subsequent updates to the block. For instance, if you set the packing density to 70%, DMS initially writes records in Output mode using 70% of the block (1428 bytes), reserving the remaining 30% of the block (612 bytes) for subsequent records written in I/O or Shared mode.

Regardless of how low you set the DPACK percentage, DMS places at least one record in each data block. Therefore, in a case where user records are 1099 characters, a DPACK field of 10% or of 100% have the same result: one record per block.

You should set the DPACK field with regard to the maximum length of the records in the file. For example, 25 80-byte records can be placed in a block. Therefore each record occupies 4% of the block. If you set the DPACK field to 96%, it will provide enough space for a one-record enlargement of each data block without block splitting.

Given the DPACK percentage, DMS calculates the number of records per block for variable length records by assuming all records to be the maximum record length. For compressed records, DMS estimates a 25% compression from the maximum record length, and assigns that many records to the file.

The DPACK field is an optional parameter of UFBGEN. If you do not specify this parameter, DMS assumes that the DPACK is 100%. When specifying the DPACK, you should express the percentage as a whole number without the percentage sign, as follows:

```
ZOOFILE   UFBGEN  FORG=INDEXED,DPACK=70,RECAREA=ZOOREC
```

If you expect the growth of the file to be slow but extensive, avoid setting the DPACK to a low percentage, because this results in degraded performance. For example, if you expect a file to slowly enlarge by 90% (which would normally require a DPACK of 10%), it is preferable to set the DPACK to a moderate percentage (say 80%), and schedule the file to be copied each time it grows 20%, using the COPY utility with REORG=YES, DPACK=80, IPACK=95. This strategy minimizes wasted space because at any one time the system reserves a maximum of only 20% for file expansion. Running the COPY utility with the REORG=YES option fixes block splits, balances block allocations, and re-establishes the 20% growth space.

9.2.4 The IPACK Field

The IPACK, or index packing density field provides the initial allocation percentage for all primary index blocks on all levels. The minimum number of table entries per index block is two; regardless of how low you set the IPACK percentage, two table entries are placed in each index block.

The IPACK field does not affect the alternate index trees, which always maintain a packing density of 100%. Lacking a packing density field, the early updates of an alternate indexed file result in block splits of the pseudo-record index blocks. After the pseudo-record blocks have been split, considerable space remains in each split block, and the situation stabilizes. The degradation in performance caused by these block splits is not serious, because the alternate index tree blocks are usually at the end of the file where the available blocks for block splits usually reside.

Setting the IPACK Field

If you have set the DPACK field to 50%, and the actual growth of the file never exceeds 50%, then the number of data blocks should never increase. The data blocks become fuller, but (assuming even growth), they never reach capacity and require a block split. In this hypothetical case, you could set the IPACK field to 100%.

In reality, however, you should not set the IPACK field to 100% in the above example. Some data block splits are inevitable, because some blocks will become filled to capacity, while others will still have space for more records. Furthermore, in the above example, as the percentage of actual file growth reaches and exceeds 50%, performance on the file would begin to degrade fairly rapidly, due to index block splitting.

Therefore, you should set the IPACK field to accommodate some data block splitting. The IPACK percentage can be calculated as follows:

$$100 - \left[\frac{[PK + 3] \times 100}{2043} \text{ rounded up} \times BS \right]$$

where PK is the length of the primary key, and BS is the number of data block splits the index block is to accommodate. In most cases, the IPACK percentage is substantially higher than the DPACK percentage.

Like the DPACK, the IPACK is set as an optional parameter to UFBGEN. You should only define an IPACK field if FORG=INDEXED.

```
ZOOFILE  UFBGEN  FORG=INDEXED,IPACK=90,DPACK=60
```

9.3 BUFFERING

A buffer is a pre-established area for receiving and temporarily holding data. Buffers are established to minimize the number of I/O operations that the system needs to perform to retrieve data from a file. Through the use of buffers, the system can transfer 2K blocks, or even larger units of data in one operation between your Segment 2 space and the data storage device. DMS performs all disk I/O operations in 2K byte block units. Tape blocks and buffers can be larger than 2K bytes.

RAM and BAM automatically provide buffering. The number of 2K buffer blocks provided depends on the type of file being accessed: a consecutive file, or a file of any type opened in BAM, is assigned one buffer; a relative file or an indexed file is assigned two buffers; and an alternate indexed file is assigned three buffers. One indexed file buffer holds the root index block and the other indexed file buffer holds the block of data currently being accessed.

If you read an indexed file sequentially, DMS uses the data buffer block for anticipatory buffer priming. When you issue a READ NEXT on the last record in a block of records, DMS assumes that the next function request against that file will also be a READ NEXT, and moves the next sequential block into the buffer before a record in that block is actually requested. Anticipatory buffer priming speeds performance by reducing the I/O wait time. A READ HOLD statement curtails anticipatory buffer priming.

9.3.1 Large Buffer Strategy

You can use the large buffer strategy for consecutive files and relative files to increase the size of the buffer. Ordinarily, DMS uses a 2K byte buffer to copy a block of data from the file into your Segment 2 space (see Chapter 6). You can increase the size of this file buffer to 18K (nine blocks), in increments of 2K. This is known as the large buffer strategy.

Larger buffers limit the number of I/O operations to the data file by copying several blocks in a single I/O operation. I/O operations tend to be time-consuming; reducing their number speeds file processing. The large buffer strategy copies up to nine adjacent file blocks, making this strategy useful in the sequential processing of files.

You establish a large buffer in UFBGEN by setting BUFSIZE equal to some multiple of 2048.

```
ZOOFILE  UFBGEN   FORG=CONSEC,BUFSIZE=4096
```

Indexed files use multiple buffers, rather than a single large buffer. DMS provides a buffer pooling strategy for the multiple 2K buffers of indexed files. You cannot specify a BUFSIZE of greater than 2048 bytes for indexed file in I/O or Shared mode. You can, however, specify a large buffer for an indexed file opened for sequential reading in Input mode.

9.3.2 Buffer Pooling Strategy

You can establish a pool of buffer blocks in Segment 2 for indexed or alternate indexed files. This buffer pool functions as a rapid access storage area for data file blocks that are frequently read (such as the high-level blocks of index trees). When your program requests a block of data, DMS checks a Buffer Control Table (BCT) to determine if the requested block is in the buffer pool. If the block is in the pool, DMS accesses the copy of the block in the pool. If the block is not in the pool, DMS locates the data block in the data file and copies it into the buffer pool. Accessing a block from a buffer pool is more efficient than accessing a block from a data file.

You can provide buffer pooling for indexed files opened in Input and I/O modes. DMS automatically provides buffer pooling for all files opened in Shared mode. Shared files use the Sharer's buffer pool. You establish the size of the Sharer's buffer pool when defining the system parameters using the GENEDIT utility. GENEDIT is described in the VS System Administrator's Reference.

Establishing a Buffer Pool

You establish a buffer pool in UFBGEN by setting parameters POOL=YES and BCT=name. The BCT (Buffer Control Table) name references a BCTGEN statement, which takes a single parameter, NBUF. NBUF specifies the number of buffer blocks that the system assigns to the table; it can be any value from 3 to 60. See Chapter 4 of the VS Operating System Services.

Example 9-1. Creating a Buffer Pool

```
ZOOFILE  UFBGEN   FORG=INDEXED,POOL=YES,BCT=ZOOPOOL
.
.
.
ZOOPOOL  BCTGEN   NBUF=60
```

How Buffer Pooling Works

When you open an indexed file for which you have specified buffer pooling, DMS established a buffer pool for that file in your Segment 2 area. DMS also establishes a Buffer Control Table (BCT) that facilitates rapid location of a block within the buffer pool.

When you access an indexed file, all of the blocks used to perform that access are copied into the buffer pool. For example, if you perform a READ KEYED, the root index block, one or more lower-level index blocks, and the data block are all copied into the buffer pool. The buffer pool retains these blocks following the access operation, so that another access request requiring the same file blocks can access them from the buffer pool, rather than having to locate these blocks in the file.

DMS initially places data blocks in the buffer pool in the order received, and lists their locations and types (root block, index block, data block) in the BCT. The BCT requires 56 bytes per buffer to list this information. Once the buffer pool is full, DMS uses a least-recently-used algorithm to replace data blocks in the buffer. This algorithm preferentially retains blocks in the buffer in the following order: alternate and primary root blocks, alternate and primary index blocks, data blocks read with HOLD, data blocks read no-hold, and alternate index pseudo-record blocks. DMS removes a block that has not been recently accessed by overwriting it or by copying it out to the file, thus creating space for a new block in the pool.

You can check the efficiency of a buffer pool by inspecting the Program Completion Report (PF3) from the Command Processor, after running the program. A Buffer Pool Statistics (PF2) option indicates the efficiency of a buffer in terms of hits and misses. DMS records a hit when it searches for a block and locates it in the buffer pool. A miss is recorded when DMS must perform an I/O operation on the data file to find a data or index block. Initially, DMS records many successive misses until the buffer pool has been filled; subtract the number of buffer blocks from the misses total before comparing the two figures. You can use the ratio of hits to misses to optimize buffer pool performance. For example, if the number of hits is less than the number of misses, you should enlarge the buffer pool in most cases.

You can, and in many cases should, use a single buffer pool for several indexed files. Example 9-2 demonstrates multiple files sharing a buffer pool.

Example 9-2. Two Files Sharing a Buffer Pool

```
FILEONE  UFBGEN  FORG=INDEXED,POOL=YES,BCT=OURPOOL
           .
           .
           .
FILETWO  UFBGEN  FORG=INDEXED,POOL=YES,BCT=OURPOOL
           .
           .
           .
OURPOOL  BCTGEN  NBUF=60
```

CHAPTER 10
DATA ACCESS METHODS

10.1 THE THREE ACCESS METHODS

DMS supports three access methods: Record Access Method (RAM), Block Access Method (BAM), and Physical Access Method (PAM). You select the method of access by means of an UFBGEN parameter that the system checks before opening a file. All three access methods can be used on any file. The access method you select determines the amount of DMS file support; file support not provided by the access method must be supplied by the user's program.

Both disk and tape devices support all three access methods. BAM and PAM are described in this chapter in terms of the 2K fixed block size required for disk files. Tape files can use these 2K units, or other block size units. Refer to Chapter 12 for further information on tape block sizes.

Selection of an access method generally depends upon the intended unit of data transfer and the level of DMS support desired. RAM is the most commonly used access method, because it provides the most complete DMS support. Only RAM supports access to logical records, and the creation or use of index trees. BAM reads and writes disk data in 2K block units only. Data transfer and copying in block units is considerably faster than in single record units. BAM reduces DMS overhead by not performing record blocking and deblocking. In addition, BAM offers the large buffer option for performing I/O to up to nine 2K buffer blocks. PAM offers the greatest flexibility and least DMS support of the three access methods. It allows you to transfer data in multiples of 2K up to 18K, and to establish asynchronous processing and specialized buffering strategies for the application. Use PAM when you want to minimize data movement or when you need a flexible user-supported buffering scheme.

10.2 RECORD ACCESS METHOD (RAM)

RAM provides the highest degree of file support. RAM is the default in access method selection; if no access method parameter is coded in UFBGEN, the file will be accessed in RAM. Unless there is a specific reason to use another access method, you should open all files in RAM. Many DMS functions, such as creation and accessing of data by index trees, location of data records within a block, compression, and file sharing are only supported in RAM. RAM is the only access method accessible from a high-level language -- you must access BAM and PAM in Assembly language. Unless otherwise noted, discussion elsewhere in this book assumes use of the Record Access Method.

Appendix B contains tables of RAM function requests available in Assembly language. For further details on RAM function request support, refer to Chapter 7.

10.3 BLOCK ACCESS METHOD (BAM)

BAM allows for block-level processing of disk files, permitting you to access, manipulate, and create files in an organization independent manner. Processing a file in block units is much faster than processing record-by-record, but possible applications of BAM are limited by its inability to access records within a block. BAM is used primarily for copying files; VS system utilities, such as COPY (with REORG=NO) and BACKUP, use BAM for fast file copy operations.

Select this access method by setting UFBF1BAM before opening the file (the BAM=YES parameter in UFBGEN). The unit of data transfer under BAM is a 2K-byte physical disk block; you can access files by relative block number (from 1) using either random or sequential methods. Data must be aligned on page boundaries. Under BAM, user programs can block or deblock the records in a file, and specify the size of the buffer to be allocated (UFBBUFSIZE). DMS automatically performs anticipatory buffer priming for BAM. See Chapter 15 for further details on anticipatory buffer priming.

The processing modes available under BAM are similar to those available under RAM: Input mode, Output mode, I/O mode, and Extend mode. Shared mode and record deletion are not supported. Before issuing a function request, the user program must specify the relative block number to be processed.

10.3.1 UFBGEN Coding for BAM

The User File Block (UFB) must be established in the STATIC (data storage) section of the program prior to opening a data file. The easiest method of establishing UFB values is by coding a UFBGEN macroinstruction. Further details on the UFB, UFBGEN and its parameters are found in Chapter 6.

Every file opened in BAM must contain the BAM=YES parameter in its UFBGEN. All files opened in BAM must also specify the following parameters that allow DMS to identify the file: FILENAME, LIBRARY, and VOLUME. You can supply these values either through UFBGEN, or at runtime by typing values to a workstation GETPARM screen. In addition, a file to be created in BAM must contain the following additional parameters: FORG (file organization), RECSIZE (record size), and either NRECS (number of records), or BLKAL and NBLKS (number of blocks). Each file definition should also include a RECAREA parameter, unless you perform Read operations without using the user record area (e.g., READ NODATA).

Typical UFBGENs for an input file and an output file (to be created with a length of ten blocks) are shown in Example 10-1.

Example 10-1. BAM File Definition Parameters

<u>Input File</u>:

```
                STATIC
                UFB     NODSECT
                ORG     UFBBEGIN
        INFILE  UFBGEN   BAM=YES,FORG=ANY,FILENAME=OLDFILE,             X
                         LIBRARY=ZOOLIB,VOLSER=ZOOVOL,RECAREA=INREC
                .
                .
                .
        INREC   DS   CL2048
```

<u>Output File</u>:

```
                STATIC
                UFB     NODSECT
                ORG     UFBBEGIN
        OUTFILE UFBGEN   BAM=YES,FORG=CONSEC,FILENAME=NEWFILE,          X
                         LIBRARY=ZOOLIB,VOLSER=ZOOVOL,RECAREA=OUTREC,   X
                         RECSIZE=2048,NRECS=10,BLKAL=YES,NBLKS=10
                .
                .
                .
        OUTREC  DS   CL2048
```

In Example 10-1, record areas are defined for the Input and Output files. In the sample BAM program shown in Appendix E, record areas are not defined, because Read operations are performed NODATA. The sample BAM program in Appendix E uses the file buffer, rather than a user record area, for storing the current record.

<u>BAM UFBGEN Parameters</u>:

FORG — The file organization. When you create a file, you must specify whether the file is indexed (FORG=INDEXED), consecutive (FORG=CONSEC), or relative (FORG=REL). When you open an existing file of any file structure, you can specify FORG=ANY. The Open operation automatically supplies the correct file organization to the file's UFB.

RECAREA	Same as RECAREA under RAM, except that the size of the field referenced by RECAREA must be one data block (2048 bytes), and the area addressed by RECAREA must be aligned on a page boundary. When using the READ NODATA function request, you can set the RECAREA equal to the file's buffer area, UFBBUFADR.
PRNAME	Same as PRNAME under RAM. A recommended optional parameter for all files.
FILENAME	Same as FILENAME under RAM. Required parameter for all files.
LIBRARY	Same as LIBRARY under RAM. Required parameter for all files.
VOLSER	Same as VOLSER under RAM. Required parameter for all files.
KEYAREA	Used for locating blocks within files. The KEYAREA=name field references a four-byte field containing the block number of the block to be read (from 1). An optional parameter.
BLKSIZE	An optional parameter used to specify the block size for both input and output files. You specify the block size for a disk in UFBGEN as BLKSIZE=2048. See Chapter 12 for the use of this field in magnetic tape access.
VLEN	The variable length record parameter is a required parameter when creating a file containing variable length records. Values are YES and NO.
COMP	The record compression parameter is a required parameter when creating a file containing compressed records. Values are YES and NO. If COMP=YES, you must also set VLEN=YES.
RECSIZE	The record size field is a required UFBGEN parameter for BAM disk file creation. The value for this field must be the actual logical record size. Although BAM writes only in physical block units, DMS can use the record size to estimate the space requirements for a new disk file created in BAM. As part of the Open operation, DMS sets UFBRECSIZE to 2048, and places the maximum record size in UFBLRECSAVE.
NRECS	You must specify either the number of records parameter or the BLKAL (block allocation) and NBLKS (number of blocks) parameters for files created using BAM. You set RECSIZE to the logical record size and NRECS to the number of logical records to be written to the file. The values of these two parameters determine the space allocation for file creation. You must change the value of the NRECS field before closing a file if you have added or deleted records to the file.

BLKAL	If you do not specify a value for the NRECS parameter, you must specify a value for the block allocation parameter in UFBGEN for files created using BAM. You should specify BLKAL=YES.
NBLKS	If you specified BLKAL=YES, you must specify the number of blocks to be allocated for files created using BAM. The NBLKS value enables DMS to allocate the primary allocation. The value of NBLKS should be the actual number of blocks to be placed in the file.

10.3.2 BAM Function Requests

BAM supports the same function requests supported under RAM, except DELETE. A READ function request copies an entire 2K block into the user record area; a WRITE or REWRITE copies a 2K block out to a data file. BAM function requests are shown in Table 10-1.

Table 10-1. BAM Function Requests and Their Modifiers

	Input Mode	Output Mode	I/O Mode	Extend Mode	Shared Mode
READ	no mod REL NODATA		no mod HOLD REL NODATA		
WRITE		no mod	no mod	no mod	
REWRITE			no mod		
START		EXTEND OUTPUT IO		EXTEND OUTPUT IO	
DELETE					

READ Function Request Modifiers for BAM

no mod	Causes DMS to read the next block in the file into the user record area. When you issue a READ after opening a file, DMS reads the first block in the file (relative Block 1).
REL	When you issue a READ REL, DMS calculates the location of the block with the specified relative block number and reads that block into the user record area. The first block in the file is relative block number 1. You specify the relative block number in the four-byte area addressed by the KEYAREA field of the User File Block as shown in Example 10-2.

Example 10-2. Use of the READ REL Function Request in BAM

```
                CODE
                MVC     BLOCKNO,=F'5'
                READ    REL,UFB=ZOOFILE
                .
                .
                .
                STATIC
       ZOOFILE  UFBGEN  BAM=YES,KEYAREA=BLOCKNO
       BLOCKNO  DS      F
```

HOLD READ HOLD must be used in I/O mode to hold a data block in the user record area. You can write the block back to the data file using a REWRITE function request (see Chapter 7).

NODATA A READ NODATA places the requested file data in the buffer area, but does not copy the specified block into the user record area. Instead, the block is retained in the DMS buffer, and the address of the block in the buffer is placed in Register 1. If the file is compressed, the data stored in the buffer is in compressed format. A READ NODATA does not update UFBRECSIZE.

You can use READ NODATA with any file or record structure. NODATA can be used by itself, or as part of a multiple modifier with REL or HOLD.

Example 10-3 shows one use of READ NODATA in BAM. The example equates the input file's buffer area with the output file's user record area. This eliminates a data transfer operation and speeds the copying of the block in BAM. A complete program using the READ NODATA function request is shown in Appendix E.

Example 10-3. Use of the READ NODATA Function Request in BAM

```
              MVC     UFBORECAREA,UFBIBUFADR
              READ    NODATA,UFB=INFILE
              WRITE   UFB=OUTFILE
              .
              .
              .
              STATIC
     INFILE   UFBGEN  MODE=IN,BAM=YES,DEVCLASS=DISK,
     OUTFILE  UFBGEN  MODE=OUT,BAM=YES,DEVCLASS=DISK
```

WRITE Function Request for BAM

You use the WRITE function request in BAM to write a 2K block of data from the user record area to an output file. WRITE takes no modifiers. It requires the UFB operand, and supports the optional COND operand for constructing a write on condition. DMS does not take the UFBEODAD error return for WRITE function requests under BAM.

REWRITE Function Request for BAM

Use the REWRITE function request in BAM to write a block of data from the user record back to the original input file in I/O mode. You use REWRITE when you have opened a file in I/O mode, and read the block using a READ HOLD. REWRITE takes no modifiers, except the UFB operand designating the file, and an optional COND operand for rewrite on condition.

Example 10-4. Use of the REWRITE Function Request in BAM

```
OPEN    UFB=IOFILE,MODE=IO
READ    HOLD,UFB=IOFILE
REWRITE UFB=IOFILE
```

START Function Request for BAM

Use the START function request to switch processing modes in BAM. This function is available in Output or Extend modes. The modifiers allowed are OUTPUT, EXTEND and IO. DMS cannot take an UFBEODAD error return when switching modes using START. In the following example, you open ZOOFILE in Output mode, but then switch the mode using the START function request to I/O mode and process the file in I/O mode:

Example 10-5. Use of the START IO Function Request in BAM

```
OPEN    UFB=ZOOFILE,MODE=OUTPUT
WRITE   UFB=ZOOFILE
START   IO,UFB=ZOOFILE
READ    HOLD,UFB=ZOOFILE
REWRITE UFB=ZOOFILE
```

10.3.3 Closing a File in BAM

Before closing a BAM file opened in Output, Extend, or I/O modes, you must update the fields in the UFB that specify the current size of the file. You must update:

UFBNRECS	the number of records in the file.
UFBEREC	For relative files and consecutive files containing fixed length records, EREC is the relative number of the final record within the E-Block. For example, if the last data record in a relative file is in the fifth record slot in E-Block, the value of EREC is 5. For variable length consecutive records, the value of EREC is 1, unless the file is a null file. For indexed files, the value of EREC is the number of levels of index blocks in the file.
UFBEBLK	the relative block number (from 0) of the highest-numbered block that contains data.

These UFB fields are automatically updated as part of the Close operation in RAM. In BAM you must manually update these fields before invoking a Close operation. When copying an entire file in BAM, you can update these fields of the output file by copying the values of the corresponding input file parameters, as shown in the BAM program example in Appendix E.

10.4 PHYSICAL ACCESS METHOD (PAM)

Physical Access Method (PAM) is the lowest level DMS support. It gives you the most control of physical I/O and buffering. However, neither the buffering capabilities of BAM, nor the logical record processing of RAM are supported under Physical Access Method.

Under PAM, your program must handle blocking and deblocking, and establish all buffer areas. You can use PAM to implement a user-defined buffering strategy. PAM itself does no buffering; all buffers must be defined in the user program. You can use PAM for devices other than disk, that require block sizes larger than 2K.

You can use PAM for initiating asynchronous I/O requests and waiting for their completion. You issue a physical I/O operation request in PAM and then issue a START WAIT function request to wait for that operation's completion. Your program can carry on asynchronous processing while awaiting the completion of an I/O operation.

10.4.1 UFBGEN Coding for PAM

When you open a file in PAM, you must specify the parameter PAM=YES in the file's UFBGEN. You must also specify the FORG, PRNAME, FILENAME, LIBRARY and VOLSER for files accessed using PAM. In addition, when you create a file using PAM, you must specify the RECSIZE, and either NRECS (the number of records to allocate space for), or BLKAL=YES (block allocation) and NBLKS (the number of blocks to allocate).

Example 10-6 shows a typical input file read using PAM, and a typical output file created using PAM.

Example 10-6. PAM File Definition Parameters

```
INFILE   UFBGEN  FORG=ANY,PAM=YES,PRNAME=IN,KEYAREA=BLOCKNO,       X
                 FILENAME=ZOOIN,LIBRARY=ZOOLIB,VOLSER=SYSTEM,      X
                 BLKSIZE=2048
BLOCKNO DC       F'7'


OUTFILE  UFBGEN  FORG=CONSEC,PAM=YES,PRNAME=OUT,                   X
                 FILENAME=ZOOOUT,LIBRARY=ZOOLIB,VOLSER=SYSTEM,     X
                 RECSIZE=80,BLKAL=YES,NBLKS=10,BLKSIZE=2048
```

PAM UFBGEN Parameters

FORG	PAM reads and writes consecutive parts of a file, independent of the internal structure of the file. For this reason, you should always set FORG=CONSEC for an output file created using PAM. You can specify an input file as FORG=ANY. FORG is a mandatory parameter.
PRNAME	Same as RAM files. A recommended optional parameter.
FILENAME	Same as RAM files. A required parameter for input files.
LIBRARY	Same as RAM files. A required parameter for input files.
VOLSER	Same as RAM files. A required parameter for input files.
KEYAREA	Used for locating a block within an input data file. The KEYAREA=name field addresses an area that contains the relative block number of the block to be read (from 0). An optional parameter.
BLKSIZE	You must specify the block size for both input and output files. This is usually done by storing the block size in the UFBBLKSIZE field in the CODE section of the program. You can specify block size in UFBGEN as well: BLKSIZE=2048. For use of this field in magnetic tape access, see Chapter 12. BLKSIZE is an optional parameter.
RECSIZE	The record size field is a required UFBGEN parameter for PAM file creation. The value for this field must be a multiple of 2048, because PAM only writes in block or multiblock units.

NRECS	You must specify either the number of records parameter or the BLKAL (block allocation) and NBLKS (number of blocks) parameters for files created using PAM. You set RECSIZE to the logical record size and NRECS to the number of logical records to be written to the file. These two parameters values determine the space allocation for file creation.
BLKAL	If you do not specify a value for the NRECS parameter, you must specify a value for the block allocation parameter in UFBGEN for files created using PAM. You should specify BLKAL=YES.
NBLKS	If you specified BLKAL=YES, you must specify the number of blocks to be allocated for files created using PAM. The NBLKS value enables DMS to allocate the primary allocation. The value of NBLKS should be the actual number of blocks to be placed in the file.

10.4.2 PAM Function Requests

PAM supports three access modes (Input mode, Output mode, and I/O mode), and four function requests (READ, WRITE, REWRITE, and START). The Extend and Shared modes are not supported in PAM. Use of the START WAIT function request is mandatory in PAM file access.

Table 10-2. PAM Function Requests and Their Modifiers

	Input Mode	Output Mode	I/O Mode	Extend Mode	Shared Mode
READ	no mod		no mod		
WRITE		no mod	no mod		
REWRITE			no mod		
START	WAIT	WAIT EXTEND OUTPUT IO	WAIT		
DELETE					

The READ, WRITE and REWRITE function requests take no modifiers. Each function request requires a UFB name operand to identify the file. You can also code a COND operand to specify conditional execution of a function request.

The READ Function Request Under PAM

The READ function request takes no modifiers. However, an unmodified READ under PAM is different from a READ under RAM or BAM. A READ function request in PAM always requests a random read operation. Therefore, it is necessary to supply a relative block number (from 0) to the area addressed by KEYAREA before issuing each READ function request. When accessing a file in I/O mode, you issue a READ with no modifier, rather than a READ HOLD. Use REWRITE in I/O mode to copy the block(s) back to the file.

The START Function Request Under PAM

The START function request, with the WAIT modifier, is used to synchronize the I/O processing of READ, WRITE, or REWRITE function requests used in PAM. You must place a START WAIT function request between any two function requests that require I/O processing on the same file. A START WAIT suspends program execution to allow the previous input or output operation to complete before executing the next instruction. START WAIT function requests are required to prevent an operation from overwriting the data placed in a buffer by the previous operation. You can minimize the effect of these START WAIT function requests by establishing separate buffers for each file being accessed.

You can use the START WAIT statement to perform simultaneous I/O transfers to several devices. A START WAIT function request waits for the completion of the I/O operation on the file specified in the UFB operand of the START WAIT.

Example 10-7. Use of the START WAIT Function Request in PAM

```
         OPEN     UFB=INFILE,MODE=INPUT
         OPEN     UFB=OUTFILE,MODE=OUTPUT
         READ     UFB=INFILE
TOP      START    WAIT,UFB=INFILE
         WRITE    UFB=OUTFILE
         READ     UFB=INFILE
         .
         .
         START    WAIT,UFB=OUTFILE
         .
         .
LOOP     B        TOP
```

In Example 10-7, the program uses two buffers to speed processing of records. Within the processing loop, the program issues a WRITE on the block already in the buffer of one file and a READ to place the next block in the buffer belonging to the other file, so that two I/O operations are occurring concurrently. A START WAIT causes program execution to wait until the first of these two operations completes. After the first I/O operation completes, the loop is taken and the program waits for the second I/O operation to complete.

You can also use the START command to change the processing mode. A file opened in Output mode can be changed to I/O or Extend mode by issuing a START command with the appropriate modifier, as shown in Example 10-8.

Example 10-8. Use of the START EXTEND Function Request in PAM

```
OPEN     UFB=INFILE,MODE=INPUT
OPEN     UFB=OUTFILE,MODE=OUTPUT
START    EXTEND,UFB=OUTFILE
READ     UFB=INFILE
```

Note that use of START for mode switching differs from mode switching in RAM. In PAM, you are only able to switch modes if the file was originally opened in Output mode. DMS checks the mode the file was opened in, not the current mode of the file.

10.4.3 Establishing Buffers for PAM

Every program that performs I/O operations using PAM must acquire at least one 2K byte buffer. In Assembly language, you code a GETBUF command prior to opening the data files to acquire a buffer block. After closing all files using the buffer block, you should code a FREEBUF command to release each buffer. See the VS Operating System Services manual, Chapter 4 for further details.

Example 10-9. Establishing and Releasing Buffers in PAM

```
        CODE
        GETBUF
        POPM      0,R5,R6
        LTR       R5,R5
        BNZ       EXIT
        OPEN      UFB=ZOOFILE
        .
        .
        .
        CLOSE     UFB=ZOOFILE
        FREEBUF   BUFLOC=(R6)
EXIT    RETURN
```

File I/O processing under PAM can be speeded greatly by creating more than one buffer, and using these multiple buffers alternately for I/O operations.

10.4.4 Establishing the PAM Record Area on a 2K Boundary

In order to input or output a record from a file using PAM, you must align the beginning of the record with a 2K block boundary. This is because in PAM the record area is the same as the buffer. Using the register containing the buffer address, you should perform the alignment for all files prior to attempting to read or write those files. For the file buffer shown in Example 10-9, you perform record area alignment as shown in Example 10-10.

Example 10-10. Record Area Alignment in PAM

```
OPEN     UFB=ZOOFILE
ST       R6,UFBRECAREA
READ     UFB=ZOOFILE
```

You should align the record area for each file accessed. If you have assigned a SUFFIX to a file's UFB, specify the suffix in UFBRECAREA and UFBBEGIN.

10.4.5 Specifying the Block Size in PAM

When DMS reads or writes a block in PAM, it resets the UFBBLKSIZE value to the amount of data actually transferred. Physical media considerations may result in an occasional transfer of less than the amount of data originally specified in UFBBLKSIZE. In order to maintain a block size of 2048 bytes (or a multiple of 2048), you must re-establish the block size in the file's UFB prior to performing a READ, WRITE or REWRITE operation on that file. You first load the block size into a register, then store the register value in each file's UFBBLKSIZE field, as shown in Example 10-11. Note the use of suffixes ("A" and "Z") to distinguish fields belonging to different UFBs.

Example 10-11. Establishing Block Size in PAM

```
         LA       R7,2048
TOP      STH      R7,UFBABLKSIZE
         READ     UFB=AZOOFILE
         .
         .
         .
         STH      R7,UFBZBLKSIZE
         WRITE    UFB=ZOONEWFILE
LOOP     B        TOP
```

Every input or output function request should be preceded by a block size store operation. Make sure to include the store halfword (STH) operation in the processing loop so that the block size in reinitialized before each READ operation, as shown is Example 10-11. If you use a suffix field to identify a file's UFB, the file's UFBBLKSIZE field name should include the suffix.

10.4.6 Closing a File in PAM

Before closing a PAM file opened in Output or I/O modes, you must update the fields in the UFB that specify the current size of the file. You must update:

UFBNRECS	the number of records in the file.
UFBEREC	For relative files and consecutive files containing fixed length records, EREC is the relative number of the final record within the E-Block. For example, if the last data record in a relative file is in the fifth record slot in E-Block, the value of EREC is 5. For variable length consecutive records, the value of EREC is 1, unless the file is a null file. For indexed files, the value of EREC is the number of levels of index blocks in the file.
UFBEBLK	the relative block number (from 0) of the highest-numbered block that contains data. EBLK is only required when the file was opened in I/O mode.

These UFB fields are automatically updated as part of the Close operation in RAM. In PAM you must manually update these fields before invoking a Close operation. When copying an entire file in PAM, you can update these fields of the output file by copying the values of the corresponding input file parameters.

PART III

Other Device and File Types

CHAPTER 11
INTERACTIVE WORKSTATION DMS

11.1 THE WORKSTATION SCREEN AS A DATA FILE

The VS Data Management System treats the workstation screen display as a data file. Interactive DMS provides you with several methods of formatting and dynamically modifying the contents of this workstation screen data file. The workstation screen is a modifiable consecutive file that consists of a single record containing a maximum of 1924 bytes. Of these 1924 bytes, 1920 bytes represent the visible screen positions known as the mapping area: 24 rows and 80 columns. The mapping area is preceded by a mandatory 4-byte order area field, which governs screen control for data transfer. The order area bytes are not displayed on the screen. The structure of the workstation file is as shown in Figure 11-1.

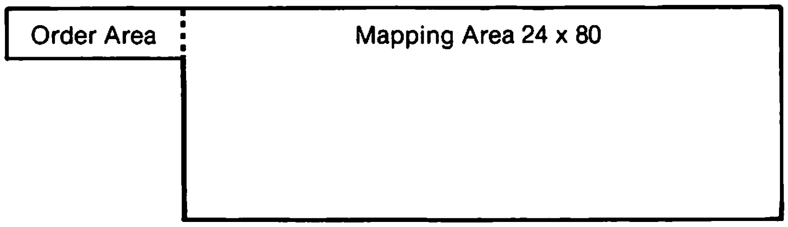


Figure 11-1. Schematic of Workstation Record

Formatting of the order area and the mapping area are described in Section 11.2.

11.1.1 Reading and Writing to the Workstation Screen

DMS workstation screen interaction is supported in BASIC, COBOL, PL/I, and RPG II. Refer to the individual language manuals for details on support in these high-level languages.

The Assembly language programmer has a choice of several different ways of transferring data to and from the workstation. The method described in this chapter is Interactive DMS, in which you access the workstation data file using the REWRITE, READ, and START function requests. This method is similar to the DMS access to disk files described in Chapters 6 and 7 of this manual. A complete sample Assembly language program for workstation interaction is provided in Appendix E.

You must establish a User File Block (UFB) in your program for each file accessed by DMS. Section 11.3 of this chapter describes the User File Block and OPEN macroinstruction coding for a workstation screen.

Interactive DMS uses three types of function requests: REWRITE, READ, and START. The use of workstation function requests differs somewhat from the use of function requests for disk file access. Because the workstation file is only one record in length, you use function requests to establish the record and to update it with another record or a modified version of the original record. The READ and REWRITE modifiers determine what part of the screen record DMS should read or write, and whether DMS should flag or modify certain fields in the process.

The REWRITE function request, which writes data from the record area to the workstation screen and issues Write Control Characters (WCC) in the order area, is described in Section 11.4. Order area coding and the WCC are also described in Section 11.4. The READ function request, which reads from the workstation screen to the record area is described in Section 11.5. The START ATTNT function request is used to record the AID completion code for the previous READ operation. The START ATTNT and AID characters are described in Section 11.6.

11.1.2 Alternatives to Interactive DMS

Other methods of writing to the workstation screen that are not described in detail in this manual include:

1. GETPARM processing, in which workstation file handling is controlled automatically by the operating system. This method allows you to format the workstation screen without generating, opening, or closing the workstation file. However, the GETPARM macroinstruction can write, at most, 18 lines of the screen and does not support all of the workstation features. The GETPARM, PUTPARM and LINKPARM macroinstructions are described in the Operating System Services manual.

2. DMS file management utilities can be used to handle many of the most common data entry and display interactions. Where applicable, the use of the CONTROL, EZFORMAT, REPORT, and DATENTRY utilities is preferable to writing a separate program for workstation I/O. These utilities are described in the VS File Management Utilities Reference.

3. You can transfer data to and from the workstation by means of the WSXIO subroutine. See the VS USERSUBS Reference for details.

11.2 FORMATTING THE WORKSTATION SCREEN

The workstation record area (RECAREA) is defined in the data section of a program. The record area begins with a 4-byte order area that is written (using the REWRITE function request) to the undisplayed screen order area. The remainder of the workstation record area is the mapping area, which the REWRITE function request maps onto the 24 rows and 80 columns of the workstation screen display.

Although the workstation record area can vary in length, its length may never be less than 4 bytes (the length of the order area) or greater than 1924 bytes. You can set the workstation record area length using the RECSIZE parameter of the UFB.

The workstation record area contains four types of information: the order area, the tabs field, field attribute characters, and field characters. You must format these items in the user record area prior to issuing a function request to display them on the screen. You can modify these values by using move commands during program execution.

11.2.1 Order Area

The order area consists of four non-displayed code bytes that govern READ and REWRITE function requests to the displayed portion of the screen. They are usually specified in hexadecimal. The four order area bytes are as follows:

Byte 0	Specifies the row of the screen that begins the mapping area. Rows are counted in hexadecimal from '01'. For example, setting this byte to hex '0C' before issuing a REWRITE function request would write from screen row 12 to the bottom of the screen.
Byte 1	Contains the Write Control Character (WCC). The WCC controls how the workstation processes a REWRITE function request. For a READ operation, set this byte to hex '00'. Write Control Characters are described in Section 11.5.
Byte 2	Specifies the column position of the cursor. The cursor only appears on the screen if you have unlocked the keyboard using the WCC (Byte 1). By setting this byte in conjunction with order area Byte 3, you can position the cursor to the first modifiable field of the screen display or to any other screen position. Columns are counted in hexadecimal from '01'.
Byte 3	Specifies the row position of the cursor. Rows are counted in hexadecimal from '01'.

11.2.2 Mapping Area

DMS writes tabs, field attribute characters, and field character designators to the displayed portion of the screen record, known as the mapping area. The up to 1920 bytes that make up the mapping area are specified as 24 rows with 80 characters per row.

Tabs Field

You can use the first ten bytes of the mapping area to specify up to ten tab positions. Each tab position specifies a column location for a tab on every line displayed on the screen. You must set tabs in ascending order in hexadecimal (hex values '01' through '50'). Following the list of tab stops, you specify the remaining bytes in the tabs field as hexadecimal zeros. Do not specify a Field Attribute Character (FAC) before the first tab position designator. If you do not want tabs, you can use the first ten characters of the mapping area as a normal workstation screen field. You cannot, however, mix tabs and data in the same workstation field.

In order to set the tabs, you must issue a REWRITE TABS function request that writes the tab stops to the workstation screen. An ordinary screen REWRITE does not set tabs. After setting the tabs, you should clear the hexadecimal characters you used to set the tabs from the mapping area. If you do not clear the mapping area, the tabs will appear as ASCII display graphic characters in the upper left corner of the workstation screen.

The tab key is only functional for a particular line if the specified tab position is within a modifiable field. Tabbing to non-modifiable positions is not supported. DMS automatically establishes the first character of all modifiable fields as a tab stop.

You can read or modify the tabs field using the READ TABS and REWRITE TABS function requests. See Sections 11.4 and 11.5 for details.

Field Attribute Characters

The Field Attribute Character (FAC) defines the screen display properties of the string of field character(s) that follow it. A FAC defines the properties of all of the field characters that follow it until either another FAC or the end of a row is encountered. You can specify any location in the mapping area, including the first ten bytes, as a FAC.

A FAC specifies the attributes of a field of up to one row in length. A field cannot extend beyond a single row of the screen; fields do not wrap around. A field may be as short as a single character. A one-character field requires two positions on the workstation screen: the Field Attribute Character and the field character.

A FAC specifies what the field's displayed characters are to look like (bright, dim, blinking, underlined, or blank (e.g., an undisplayed password field)), whether a user can modify the field, and what characters are allowed for modification (all alphanumerics, uppercase alphanumerics only, numeric only). Refer to Table 11-1 for a complete listing of FAC combinations.

By convention, bright characters are used for modifiable data, dim characters are used for protected data fields. A blinking character is alternately high-intensity (bright) and low-intensity (dim). Blinking characters are conventionally used to indicate an error, or to issue a warning. The RESET key resets all displayed blinking characters to bright characters. See Section 11.4 for further details on blinking characters.

If you do not specify a FAC in a particular row, DMS considers the row to have the default FAC specifying a low-intensity, protected field. Default field attribute characters, unlike programmer-supplied field attribute characters, do not occupy a position of the screen.

You can set a FAC in hexadecimal or binary. You can specify a FAC as a separate data field, or as part of a character string. You can quickly recognize a FAC in hexadecimal by a value of '80' or greater, and in binary by a '1' in bit location 0. FACs with values of hex 'C0' or greater, or with binary bit locations 0 and 1 set to '11', mark selected data fields for the REWRITE SELECTED and READ ALTERED function requests.

Table 11-1. Field Attribute Character (FAC) Values in Hexadecimal and Binary

				non-selected		selected	
Bright	Modify	All	No line	80	10000000	C0	11000000
Bright	Modify	Uppercase	No line	81	10000001	C1	11000001
Bright	Modify	Numeric	No line	82	10000010	C2	11000010
Bright	Protect	All	No line	84	10000100	C4	11000100
Bright	Protect	Uppercase	No line	85	10000101	C5	11000101
Bright	Protect	Numeric	No line	86	10000110	C6	11000110
Dim	Modify	All	No line	88	10001000	C8	11001000
Dim	Modify	Uppercase	No line	89	10001001	C9	11001001
Dim	Modify	Numeric	No line	8A	10001010	CA	11001010
Dim	Protect	All	No line	8C	10001100	CC	11001100
Dim	Protect	Uppercase	No line	8D	10001101	CD	11001101
Dim	Protect	Numeric	No line	8E	10001110	CE	11001110
Blink	Modify	All	No line	90	10010000	D0	11010000
Blink	Modify	Uppercase	No line	91	10010001	D1	11010001
Blink	Modify	Numeric	No line	92	10010010	D2	11010010
Blink	Protect	All	No line	94	10010100	D4	11010100
Blink	Protect	Uppercase	No line	95	10010101	D5	11010101
Blink	Protect	Numeric	No line	96	10010110	D6	11010110
Blank	Modify	All	No line	98	10011000	D8	11011000
Blank	Modify	Uppercase	No line	99	10011001	D9	11011001
Blank	Modify	Numeric	No line	9A	10011010	DA	11011010
Blank	Protect	All	No line	9C	10011100	DC	11011100
Blank	Protect	Uppercase	No line	9D	10011101	DD	11011101
Blank	Protect	Numeric	No line	9E	10011110	DE	11011110
Bright	Modify	All	Underline	A0	10100000	E0	11100000
Bright	Modify	Uppercase	Underline	A1	10100001	E1	11100001
Bright	Modify	Numeric	Underline	A2	10100010	E2	11100010
Bright	Protect	All	Underline	A4	10100100	E4	11100100
Bright	Protect	Uppercase	Underline	A5	10100101	E5	11100101
Bright	Protect	Numeric	Underline	A6	10100110	E6	11100110
Dim	Modify	All	Underline	A8	10101000	E8	11101000
Dim	Modify	Uppercase	Underline	A9	10101001	E9	11101001
Dim	Modify	Numeric	Underline	AA	10101010	EA	11101010
Dim	Protect	All	Underline	AC	10101100	EC	11101100
Dim	Protect	Uppercase	Underline	AD	10101101	ED	11101101
Dim	Protect	Numeric	Underline	AE	10101110	EE	11101110
Blink	Modify	All	Underline	B0	10110000	F0	11110000
Blink	Modify	Uppercase	Underline	B1	10110001	F1	11110001
Blink	Modify	Numeric	Underline	B2	10110010	F2	11110010
Blink	Protect	All	Underline	B4	10110100	F4	11110100
Blink	Protect	Uppercase	Underline	B5	10110101	F5	11110101
Blink	Protect	Numeric	Underline	B6	10110110	F6	11110110
Blank	Modify	All	Underline	B8	10111000	F8	11111000
Blank	Modify	Uppercase	Underline	B9	10111001	F9	11111001
Blank	Modify	Numeric	Underline	BA	10111010	FA	11111010
Blank	Protect	All	Underline	BC	10111100	FC	11111100
Blank	Protect	Uppercase	Underline	BD	10111101	FD	11111101
Blank	Protect	Numeric	Underline	BE	10111110	FE	11111110

FACs occupy a space on the workstation screen but are not displayable as they do not correspond to any ASCII character. All FACs appear on the screen as blank non-modifiable spaces. Therefore, you can use a FAC to establish a mandatory space between two data fields; for example, between an area code and a telephone number.

DMS automatically establishes FACs that precede modifiable fields as tab stops to allow you to tab immediately to the beginning of each modifiable field. You can use the tabs field to establish additional tabs within the modifiable fields (see preceding description).

Field Characters

Any character in the ASCII character set can be written from the mapping area to the screen display. You can define ASCII characters that do not correspond to keys on the workstation keyboard in the mapping area in hexadecimal. The complete ASCII character set is listed in the Quick Reference documents for BASIC or COBOL.

The pseudoblank is a field character that is used specifically for screen display. You write this character as a hex '0B'; it appears on the screen as a solid box. When used with FACs denoting modifiable fields, pseudoblanks indicate at a glance the location and length of all modifiable fields. You can copy a pseudoblank to the user record area as either a pseudoblank or a normal blank, depending on the type of READ function request you invoke.

11.2.3 A Workstation Screen Format Example

Example 11-1 defines an entire workstation screen, both the non-displayed order area and the displayed mapping area. The order area is coded on line "ORD". Row 1 contains tabs information, and Rows 2, 3, 4, and 5 contain FACs and field characters. Row 2 is a data entry area for a last name; Row 5 is a data entry area for a telephone number.

Example 11-1. A Sample Screen Record Area

```
.
.
.
                DS   0F
ZOOREC          DS   0CL1924
ORDAREA         DS   0XL4
MAPSTART        DC   X'01'
WCC             DC   X'A2'
CURCOL          DC   X'C0'
CURROW          DC   X'02'

MAPAREA         DS   0CL1920
*****   ROW 1   SET TABS  *****
TABS            DC   X'0A141E28323C46000000'
                DC   C70' '
*****   ROW 2   MODIFIABLE NAME AREA *****
                DC   CL10' '
                DC   X'80'
NAME            DS   CL12
                DC   X'8D'
                DC   CL56' '
*****   ROW 3   LABEL FOR NAME AREA *****
                DC   X'8D'
                DC   CL14' '
LABEL           DC   C'NAME'
                DC   CL61' '
*****   ROW 4   BLANK LINE *****
                DC   CL80' '
*****   ROW 5   MODIFIABLE PHONE NUMBER FIELD  *****
                DC   CL10' '
                DC   X'82'
AREAC           DC   C'617'
                DC   X'82'
PHONE1          DS   CL3
                DC   X'82'
PHONE2          DS   CL4
                DC   X'8C'
                DC   CL53' '
*****   REST OF SCREEN  *****
                DC   19CL80' '
                END
```

The following is a description of the screen display generated in Example 11-1:

Order Area — The four order area bytes are not displayed on the workstation screen. Bytes two and three of the order area automatically position the cursor to the first character of the name field in Row 2.

Row 1	Example 11-1 establishes seven tabs in Row 1. Their column locations are written in the mapping area in hexadecimal. After setting these tabs, you should clear the mapping area. Only one of the tab positions, hex '14', points to a column containing modifiable screen characters (in Rows 2 and 5). This is the only functional tab stop for this screen display. The other tabs are set, but are not usable on this particular screen.
Row 2	Writes a field of twelve modifiable bright pseudoblanks to the screen display. This modifiable area begins in Column 11 of Row 2.
Row 3	Displays the word "name" in a 4-byte, non-modifiable, low-intensity field in Row 3. This label is offset to Column 15 to be centered under the pseudoblanks in Row 2.
Row 4	Is a blank line.
Row 5	Contains three modifiable fields that make up a telephone number. The first field consists of a modifiable area code with a default value of 617. This is followed by seven modifiable pseudoblanks divided into a field of three and a field of four. These two fields are separated by a non-modifiable space created by means of a redundant FAC character.

11.3 ACCESSING THE WORKSTATION SCREEN

Interactive DMS access is similar in many ways to record access of data files on disk. In interactive DMS, the VS workstation acts as a consecutive DMS file, containing one non-compressed 1924-byte record. You must open this workstation file in I/O mode using the Record Access Method (RAM). You must place parameter values in the file's User File Block (UFB) before issuing an OPEN statement to access the data file. You use the REWRITE, READ, and START function requests to perform I/O operations. After record processing is completed, you use the CLOSE macroinstruction to relinquish the data file.

11.3.1 The User File Block

Before opening the file, your program must generate a User File Block (UFB) for the workstation file. In Assembly language, you can create a UFB by coding a UFBGEN and its appropriate parameters.

The mandatory parameters for a workstation UFBGEN are PRNAME and DEVCLASS; once you have identified the device as a workstation (DEVCLASS=WS), the system takes the appropriate UFB default values; you do not have to provide values for parameters such as FORG and MODE. You code a workstation UFBGEN as shown in Example 11-2.

Example 11-2. Workstation File UFBGEN Coding

```
STAT    STATIC
        UFB       NODSECT
        ORG       UFBBEGIN
ZOOFILE UFBGEN    PRNAME=WSFILE,DEVCLASS=WS
```

You can either specify other UFB parameters, such as RECSIZE, RECAREA, and KEYAREA, in the UFBGEN, or set them in your program before accessing the file. Thus, a typical creation of a workstation UFB would be as shown in Example 11-3.

Example 11-3. Workstation RECAREA Coding

```
STAT    STATIC
        UFB      NODSECT
        ORG      UFBBEGIN
        DS       0F
SCREEN  UFBGEN PRNAME=WSFILE,DEVCLASS=WS,RECSIZE=1924,                X
               RECAREA=ZOOREC,KEYAREA=ZOOKEY
ZOOREC  DS     0F                  FULL WORD ALIGNMENT
        DC     X'01A00101'             ORDER AREA
        DC     24CL80'EXAMPLE'         MAPPING AREA
ZOOKEY  DS     F
```

UFBGEN Operands

RECSIZE You can read or write either the entire workstation record area or a portion of it as one record. You can set the record length in the RECSIZE field of the UFB. The record length can be dynamically changed during program execution. The record length in RECSIZE includes both order and mapping areas. The maximum RECSIZE is 1924, the minimum is 4. A record size of 84 is recommended for reading by rows.

Example 11-4. Workstation RECSIZE Coding

```
SCREEN    UFBGEN DEVCLASS=WS,RECSIZE=1924
```

RECAREA The RECAREA field of the UFB contains the address of the user record area; the user record area in workstation processing is the 4-byte order area followed by the (RECSIZE minus 4)-byte mapping area. The address given in RECAREA can point to a fullword aligned location in the STATIC section, as shown in Example 11-5, or to a RECAREA stored in the user's stack or heap areas.

Example 11-5. Workstation RECAREA Structure

```
SCREEN    UFBGEN DEVCLASS=WS,RECAREA=AREA,RECSIZE=84
AREA      DS  0F                  FULL WORD ALIGNMENT
          DC  X'01A00101'         ORDER AREA
          DC  CL80'EXAMPLE'       MAPPING AREA
```

KEYAREA If the KEYAREA field has a non-zero value, DMS will move the value from the rightmost byte of the 4-byte KEYAREA field to the leftmost byte (byte 0) of the order area. Byte 0 of the order area specifies the beginning row for a READ or REWRITE operation. Currently, KEYAREA is only used in COBOL to dynamically modify the order area Byte 0.

Example 11-6. Workstation KEYAREA Coding

```
SCREEN  UFBGEN  DEVCLASS=WS,RECAREA=REC,KEYAREA=KEY

          DS  0F
KEY       DC  F'19'            *Begin reads at row 19.*

REC       DC  0X1924
ORDER     DC  X'01000000'      *Begin reads at row 1. *
```

11.3.2 The OPEN and CLOSE Macroinstructions

The VS workstation, like all DMS data files, must be opened using the OPEN macroinstruction prior to reading or writing to the file, and relinquished using the CLOSE macroinstruction after file processing is completed. The VS workstation is a consecutive file that must be opened in I/O mode. I/O mode must be specified either during UFB parameter definition (UFBGEN) or in the OPEN statement, or both. A typical workstation file is opened and closed as shown in Example 11-7.

Example 11-7. Workstation OPEN and CLOSE

```
        CODE
OPEN    UFB=SCREEN,MODE=IO
REWRITE UFB=SCREEN
READ    UFB=SCREEN
.
.
.
REWRITE UFB=SCREEN
CLOSE   UFB=SCREEN
```

A READ statement is often the last function request you issue before closing a file. DMS reads the screen display into the record area; from there you can write the data to a disk or tape file.

11.4 THE REWRITE FUNCTION REQUEST

The REWRITE function request allows you to write data from the workstation record (order area and mapping area) to all or part of the workstation screen. Usually, you issue a REWRITE before you issue a screen READ. This writes the fields established in the mapping area to the screen display and sets the order area. You can use the Write Control Character (WCC), located in the order area, for a variety of workstation controls. The most common of these is unlocking the keyboard, which allows responses from the workstation operator.

Three REWRITE commands -- REWRITE, REWRITE SELECTED, and REWRITE TABS -- control what type of data is written to the workstation screen. REWRITE [no modifier] writes all types of data from the mapping area to the screen; REWRITE SELECTED writes only selected fields to the screen; and REWRITE TABS writes only the 10-byte tabs field from the mapping area to the workstation screen.

All REWRITE function requests write the four order area bytes to the workstation. The order area bytes control the cursor, lock or unlock the keyboard, and control other aspects of the workstation.

11.4.1 The REWRITE Order Area

The order area consists of four bytes, numbered left to right from zero. Each byte has an established function, as shown in Table 11-2.

Table 11-2. Order Area Byte Schema

Byte	Function
0	Row number of first line to be written or read.
1	Write Control Character (WCC)
2	Cursor column address (used only if the position cursor bit has been set in the WCC)
3	Cursor row address (used only if the position cursor bit has been set in the WCC)

Byte 0 Begin REWRITE Row Address

The row number (Byte 0) is the number of the row at which reading from or writing to the workstation begins; if the KEYAREA field of the UFB is set to zero (the usual case), you specify the row as a hexadecimal value between X'01' and X'18' (decimal 1-24). A user-specified row value of 0 or 25 or greater terminates the command with an indication of order check (File Status 34). You can also set the row number by specifying a fullword value between 1 and 24 in the KEYAREA field of the UFB. If the KEYAREA field has a non-zero value, DMS will move the value from the KEYAREA to Byte 0 of the order area. The function of Byte 0 depends on the options selected in Byte 1, the Write Control Character.

Byte 1 The Write Control Character (WCC)

The second byte of the order area is the Write Control Character (WCC). The WCC allows you to set parameters for the REWRITE function request. Table 11-3 relates each bit in the WCC to a write option; a description of each option follows the table. Bytes 2 and 3 of the order area have no meaning independent of the WCC.

Table 11-3. Write Control Character Values

Bit	Write Option (if the bit is set to 1, the specified action will occur)
0	Unlock keyboard
1	Sound alarm
2	Position cursor
3	Roll down
4	Roll up
5	Erase rest of modifiable fields
6	Erase and protect rest of screen
7	Reserved (must be 0)

Unlock keyboard — If this bit is 1, the system unlocks the keyboard after displaying the mapping area on the screen. When the keyboard is unlocked, all keys are operational for data entry.

If this bit is 0, the system will lock the keyboard before any data transmission to the workstation, locking an unlocked keyboard. If the keyboard is already locked, an attempt by the REWRITE to lock the keyboard will have no effect. If the bit is set to 0 and the keyboard is initially unlocked, the keyboard is locked upon reception of one of the communication keys (ENTER, HELP or a PF key), and the AID character (described in Section 6.11) is set to a hexadecimal 21. If the bit is zero and the keyboard is already locked, the AID character will not change.

Sound alarm

If this bit is 1, the alarm will sound before data is transmitted to the screen.

Position the cursor

If this bit is 1, and if the keyboard is unlocked, following data transfer to the screen DMS positions the cursor to the column specified in Byte 2 of the order area and row specified in Byte 3 of the order area.

If this bit is 0, the cursor will remain in its position prior to the REWRITE, regardless of the values set in Bytes 2 and 3 of the order area.

Roll down

If you set Bit 3 of the WCC to 1, the bottom line of the screen will be lost and each line above it, up to and including the row specified in Byte 0 of the order area, will be copied into the next lower line. DMS then sets the specified row to blanks and continues the REWRITE function request. An attempt to write a record larger than the amount of room cleared on the screen with the "roll down" option will result in an order check error.

Roll up

If you set Bit 4 of the WCC to 1, the row specified in Byte 0 of the order area will be lost and each line below it, up to and including the last line of the screen, will be copied into the next higher line (e.g., Line 1 will be replaced by the contents of Line 2, etc.). DMS will then set the last line to blanks and proceed with the REWRITE on the last line of the screen. An attempt to write more than one line with the "roll up" option in a single command will result in an order check.

Erase rest of modifiable fields

If you set Bit 5 to a value of 1, DMS writes pseudoblanks to all modifiable locations between the beginning of the row specified in Byte 0 of the order area and the end of the screen before transferring data to the screen.

Erase and protect rest of screen	If you set Bit 6 to a value of 1, DMS sets all locations of the screen from the row address specified in Byte 0 of the order area to the end of the screen to the Field Attribute Character '8C' (dim, protected) before it transfers data to the screen. You cannot subsequently modify any of the data appearing on the screen.

WCC Sequence of Execution

The execution order of the WCC options on a REWRITE on a parallel workstation is shown in Table 11-4. The order of Steps 3 and 4 is reversed for serial and combined workstations.

Table 11-4. Write Control Character Sequence of Execution

1. lock keyboard (optional)
2. sound alarm if specified
3. roll up or down if specified
4. erase modifiable or protect if specified
5. WRITE data
6. unlock keyboard (optional)

Bytes 2 and 3 -- Cursor Positioning Bytes

The Bytes 2 and 3 of the order area specify the row and column location at which to initially position the cursor. Usually, you should set these bytes to the row and column number of the first modifiable byte of the first screen field. Cursor positioning only takes place if you have set the cursor positioning bit in the Write Control Character (order area Byte 1).

If the value of the row specified in Byte 3 is zero, DMS will consider the value to be 1. If the value of the column specified in Byte 2 is zero, the cursor will be positioned as if a tab key had been pressed from an initial cursor position one location before the beginning of the row just written. However, if there are no modifiable fields on the rest of the screen, the cursor will be positioned to the first location in the row specified in Byte 3 and an alarm will sound.

If Byte 2 contains a column address other than 00-50 hexadecimal (0-80 decimal) or Byte 3 contains a cursor row address other than 0-18 hexadecimal (0-24 decimal), the REWRITE command will be terminated with an indication of order check (File Status '34').

11.4.2 REWRITE Function Request Modifiers

There are only three legal forms of the REWRITE function request: REWRITE, REWRITE SELECTED, and REWRITE TABS. The REWRITE function request transfers all data from the mapping area to the corresponding screen positions; REWRITE SELECTED transfers only the contents of fields with the selected field tags set in the mapping area to corresponding screen locations; and REWRITE TABS unsets all tabs and changes them to the tabs specified in the first ten bytes of the mapping area. All forms of the REWRITE instruction reset UFBFS1 to indicate either successful completion of the instruction or an order check, and place the AID character in the UFBFS2 field. The functional differences of the three forms of the command are detailed below.

REWRITE: The unmodified REWRITE function request transfers all data, unchanged, to the workstation screen, including the four order area bytes. The command functions whether or not the keyboard is locked; however, it is normally undesirable to issue a REWRITE to an unlocked keyboard, because operator keystrokes could be lost. Use of this function request is shown in Example 11-8.

Example 11-8. Workstation REWRITE Coding

```
OPEN     UFB=SCREEN,MODE=IO
REWRITE UFB=SCREEN
```

REWRITE SELECTED: This modified form of the REWRITE function request transfers to the screen all fields in the mapping area that have the selected field bit set in their Field Attribute Characters (see Table 11-1). DMS does not reset the selected field bits in the mapping area; it turns off the selected field bits at the workstation for those FACs that identify the fields that were written.

Example 11-9. Workstation REWRITE SELECTED Coding

```
REWRITE SELECTED,UFB=SCREEN
```

REWRITE TABS: This modified REWRITE function request unsets all current tabs, and writes to the workstation up to ten new tabs that are stored in the first ten bytes of the mapping area. The REWRITE function request does not write the other areas of the screen. You must specify each new tab stop in the mapping area by column number in ascending order (hexadecimal 00 through 50). A column number of 00 signals the end of the list of tab settings; DMS does not examine the contents of any subsequent bytes.

If you specify a screen file for setting tabs, its RECSIZE must be at least 14 bytes, or DMS will reject the command with an indication of a length error (File Status '97').

Example 11-10. Workstation REWRITE TABS Coding

```
OPEN     UFB=SCREEN,MODE=IO
MVC      TABS,=X'01050A10151A'
REWRITE  TABS,UFB=SCREEN
```

11.5 THE READ FUNCTION REQUEST

The READ function request retrieves data from either part or all of the workstation display (both the displayed screen characters, and the non-displayed order area bytes), and places the data in the user record area. From there, you can rewrite the screen record back to the screen, or write it to a disk or tape file.

When specifying a READ operation, you should establish four field values: the order area byte values, if different from the values specified for REWRITES; the RECSIZE; the Field Attribute Character values, and the READ function request modifier.

You can specify a READ function request that reads only a portion of the 1924-byte screen display. The portion of the workstation record read depends upon the value of Byte 0 of the order area and the value of RECSIZE coded in the UFB.

11.5.1 The READ Order Area

The leftmost of the four order area bytes, known as Byte 0, provides you with one method of reading only a portion of the screen display. This byte specifies the row at which the READ should start. If you want a full-screen READ, set this byte to 1, and set the record length to 1924. Specifying a higher value than 1 for Byte 0 indicates that only that row and rows with higher numbers (rows displayed lower on the screen) are to be read. You must specify the row number as a hexadecimal value between X'01' and X'18' (decimal 1-24); a value out of range results in an order check error (File Status '34'). You must adjust the RECSIZE to reflect the value you set in order area Byte 0. Each increment of Byte 0 above '01' should be reflected in a reduction of the maximum RECSIZE by 80 characters, as follows:

Byte 0:	Maximum RECSIZE
'01'	1924
'02'	1844
'03'	1764
.	
.	
.	

If you want full-screen READs and REWRITEs, you must establish the order area byte values for the REWRITE function request. In most cases it is not necessary to code a separate order area for READ processing.

The order area Byte 1 is not used for READ processing, and should be set to zeros.

The initial values of order area Bytes 2 and 3 are not important, because the values read for these bytes are the values established by the position of the cursor when you invoke the READ function request. Order Area Byte 2 takes the column position number of the cursor. Byte 3 of the order area takes the row position number of the cursor when you invoke the READ operation.

11.5.2 Field Attribute Characters

You specify Field Attribute Characters (FAC) in the data section of your program, immediately prior to the data field for which they supply attributes. FACs occupy a space on the workstation screen but are not displayable, as they do not correspond to any displayable ASCII character. They are coded using either hexadecimal or binary values.

A READ function request copies the FACs along with the other characters from the screen to the mapping area. If you do not supply a FAC for a screen field, DMS supplies a default FAC. These default FACs are not read to the mapping area.

You can inspect FAC characters in the mapping area using the DISPLAY utility. DISPLAY in decimal mode represents all FACs by a dot in the user record; you can identify FACs in DISPLAY hexadecimal mode by their hexadecimal values.

The execution of READ function requests depends on the FAC values of the fields read. READ function requests with certain modifiers read only those data records specified in their FACs as modified or as modifiable.

You should take note of the BLINK field attribute. Following READ function requests with certain modifiers, the fields you established as blinking either continue to blink, or are reset by DMS as bright nonblinking characters.

11.5.3 READ Function Request Modifiers

You can use three of the four workstation READ modifiers to read the entire workstation display (order and mapping areas). The fourth, READ TABS, is used to read the workstation tab positions, not screen display information. The screen display READs differ in how they handle non-modified screen fields, pseudoblanks, and blinking characters.

A non-modified screen field is a modifiable data entry field to which you have written no data. Data entry fields are usually displayed as pseudoblanks, which are represented as hexadecimal '0B'. When a data entry field is read from the screen, the pseudoblanks are either copied as is, or converted to normal blanks. The three workstation display READs are summarized in Table 11-5.

Table 11-5. Workstation READ Function Requests

	Non-modified Screen Field	Modified Screen Field	Blinking Characters	Transferred Record
READ [no modifier]	Pseudoblanks unchanged by READ	Trailing pseudoblanks unchanged	Blink	Pseudoblanks read for all screen fields
READ MOD	Pseudoblanks changed to spaces	Trailing pseudoblanks = spaces	Blink reset to bright	Pseudoblanks read as spaces
READ ALTERED	Pseudoblanks unchanged by READ	Trailing pseudoblanks = spaces	Non-mod. fields blink	Pseudoblanks read for non-mod. fields

In interactive DMS, the READ function request has four valid forms for manipulating a workstation file: READ, READ MOD, READ ALTERED, and READ TABS. The functional differences between the four forms of the Interactive DMS READ function request are detailed below.

READ — The unmodified READ function request causes the contents of the screen locations corresponding to the mapping area to be copied into the mapping area without change. Thus, all characters and field attribute characters within the defined record are read. Pseudoblank characters, blinking fields, and modified data tags are transferred to the mapping area unchanged.

Example 11-11. Workstation READ Coding

```
OPEN    UFB=SCREEN,MODE=IO
REWRITE UFB=SCREEN
READ    UFB=SCREEN
```

READ MOD

The READ function request with the MOD modifier reads the modifiable fields from the workstation screen display to corresponding locations in the mapping area. It skips over protected fields within the record, leaving the corresponding mapping area locations unchanged. It converts modifiable pseudoblank characters on the screen into blanks before transmitting them to the mapping area, and changes blinking characters to high intensity non-blinking characters.

Example 11-12. Workstation READ MOD Coding

```
OPEN     UFB=SCREEN,MODE=IO
REWRITE  UFB=SCREEN
READ MOD,UFB=SCREEN
```

READ ALTERED

The READ function request with the ALTERED modifier causes the contents of those fields that have the modified data tag (the second bit) set in their FAC to be copied into corresponding positions of the mapping area. The modified data tags in the portion of the screen read are turned off at the workstation, but remain set in the corresponding FACs in the mapping area. Pseudoblanks are converted to blanks for modified fields, but pseudoblanks and blinking characters are preserved unchanged for non-modified fields.

Example 11-13. Workstation READ ALTERED Coding

```
OPEN    UFB=SCREEN,MODE=IO
REWRITE UFB=SCREEN
READ ALTERED,UFB=SCREEN
```

READ TABS — The READ function request with the TABS modifier reads the column numbers of up to ten set tabs into the first ten bytes of the mapping area. It does not read any modifications made to workstation screen fields. The record size specified in the UFB must be at least 14 (even if you are setting fewer than 10 tabs) or DMS will terminate the operation with an order check (File Status '34').

Example 11-14. Workstation READ TABS Coding

```
OPEN     UFB=SCREEN,MODE=IO
REWRITE TABS,UFB=SCREEN
READ TABS,UFB=SCREEN
```

11.6 THE START ATTNT FUNCTION REQUEST

There is only one legal form of the START instruction for workstation files: START ATTNT. The START ATTNT function request retrieves the most recent AID character. An AID character specifies either which PF key was pressed in the previous workstation interaction or the status of the workstation (keyboard locked or unlocked). START ATTNT places the AID character in the UFB File Status 2 field (UFBFS2) and places a 0 in the UFBFS1 field of the UFB.

A READ function request also places the most recent AID character in UFBFS2. Use START ATTNT when you wish to retrieve the AID character without issuing a READ when you wish to see if the keyboard needs to be unlocked. You can examine the contents of UFBFS2 to determine the last AID character issued. Establishing addressability to examine UFB field values is described in Chapter 6.

The most common use of the START ATTNT is to interrupt a running program from the workstation, as shown in Example 11-15. If the keyboard is unlocked, pressing the ENTER key or any of the PF keys generates an AID character. The START ATTNT replaces the "Keyboard Unlocked" AID character (a blank) with the AID character for the key pressed. A running program can repeatedly issue a START ATTNT, and then check the contents of UFBFS2. If an interrupt key is pressed, START ATTNT places the AID character for that key in UFBFS2. Example 11-15 checks for the presence of an AID character 'P' (PF 16) each time the program loop is performed. It then uses this AID character value to branch to an interrupt subroutine.

Example 11-15. Workstation START ATTNT Coding

```
              MVI   UFBFS2,C'0'
       TOP    START ATTNT,UFB=ZOOFILE
              CLI   UFBFS2,C'P'
              BE    INTRUPT
              .
              .
              .
       BOTTOM B     TOP
       INTRUPT
```

11.6.1 The AID Character

The current status of the workstation, i.e., whether or not the keyboard is locked, or which of the PF or ENTER keys has been pressed, is indicated in the AID character. The UFB contains two fields reflecting file status: UFBFS1, which indicates the completion status of the DMS I/O function; and UFBFS2, which always contains the most recent AID character after a successful READ or REWRITE and can be updated at any time with the START ATTNT function request. On abnormal I/O completion, the UFBFS2 contains an error code indicating the reason for I/O termination.

The AID characters are one-character codes for the ENTER key and PF keys, as well as for certain error status conditions. ENTER is represented by an '@' sign, the lowercase PF keys (PF1 through PF16) are represented by the capital letters 'A' through 'P', and the uppercase PF keys (17 through 32) are represented by the small letters 'a' through 'p'. For a chart of AID characters, see Chapter 9 of the VS Principles of Operation.

11.7 WORKSTATION ERROR COMPLETION CODES

The file status fields, UFBFS1 and UFBFS2, are used to store two-character codes that indicate abnormal completion of workstation READ and REWRITE operations.

File Status	Meaning
C'10'	No data read during a READ ALTERED (end-of-data).
C'30'	I/O error.
C'34'	Order check.
C'95'	Invalid function or function sequence.
C'96'	Invalid data area location or alignment.
C'97'	Invalid length for device.

DMS uses the first character of the file status field, UFBFS1, to hold workstation error codes. This field is reset to ASCII zero (hex '30') by the successful completion of a READ, REWRITE or START ATTNT function request. The UFBFS1 codes and meanings are shown in Table 11-6.

Table 11-6. Workstation Error Completion Codes

UFBFS1 value	Meaning
C'0'	Successful completion.
C'1'	At end.
C'2'	Invalid key or record number.
C'3'	Permanent I/O error.
C'6'	Cancel code stored.
C'9'	Other conditions.

DMS uses the UFBFS2 field to hold the AID character following the successful completion of a function request. For further information on file status codes, refer to Chapter 14.

CHAPTER 12
DMS MAGNETIC TAPE SUPPORT

12.1 INTRODUCTION

The VS Data Management System supports the reading and writing of seven- and nine-track magnetic tapes, accessed either by physical units (BAM and PAM) or by logical records (RAM). This chapter describes DMS magnetic tape processing, using terms and examples from Assembly language. All VS high-level languages provide magnetic tape support; refer to the individual language manuals for details.

Because magnetic tape is a consecutive storage medium, a file stored on magnetic tape can only be processed as a consecutive file. A consecutive tape file can be created (written in Output mode), read (read in Input mode), or extended (additional records written to the end the file). You cannot update records within tape files. The physical limitations of magnetic tape prevent either the random locating of specific records within files, or the updating of records.

A reel of magnetic tape can contain more than one file. Alternatively, a single file can span several tape volumes. This allows for the creation of quite large files. Blocks within tape files can be up to 32K bytes, considerably larger than the 2K maximum block size for disk files.

This chapter does not attempt to completely describe tape processing on the VS. The physical mounting of tape volumes and other hardware aspects of tape and tape drives are described in the VS Systems Operation Guide, Chapter 15; logically mounting and dismounting tapes is described in the VS Programmer's Introduction, Chapter 3. A blank magnetic tape can be formatted using the TAPEINIT utility; a tape can be copied from another tape volume or to a disk volume by means of the TAPECOPY utility. These utilities are described in the VS System Utilities Reference, Chapters 7 and 17. Tape track formats, the physical I/O of tape, and resulting IOP error codes are the topics of Chapter 12 of the VS Principles of Operation.

12.2 DMS-SUPPORTED TAPE FORMATS

DMS supports a variety of tape formats, facilitating the transfer of data between the VS and other computers. Both seven- and nine-track tape are supported; however, DMS support is not provided for the 2529V one-track cartridge tape drive. An Assembly language program accessing a seven-track tape must specify TRACK7=YES as a UFBGEN parameter. Similar track operands are available for high-level languages (with the exception of Fortran 66, which does not currently support seven-track tape).

DMS uses parity checking to prevent errors in reading magnetic tape. DMS automatically generates parity bit values when writing to tape, and performs parity checks on all data read from tape. Wang nine-track tape drives support even parity. You can establish odd or even parity for seven-track tapes by using the PARITY parameter of UFBGEN. Odd parity is desirable if explicit blank information (hex '00') is to be encoded on seven-track tape. When reading a tape, the parity option specified in the UFBGEN must agree with the parity of the tape volume.

DMS supports two densities of seven-track tape: 556 and 800 bits per inch (BPI). It supports three densities of nine-track tape: 800, 1600, and 6250 BPI. Use the DEN parameter of UFBGEN to set the tape density.

NOTE

The tape density selected in the user program must be a density supported by the tape drive.

A tape can be labeled or unlabelled. A labeled tape contains a volume label at the beginning of the tape volume and file labels at the beginning of each file on the tape. Wang supports three UFBGEN tape label options: Wang ANSI-standard labels (AL), IBM-standard labels (IL), and non-labelled tapes (NL). Seven-track tapes are always non-labelled. ANSI and IBM labels are similar in content and organization; the principal difference is that ANSI labels are written in ASCII, whereas IBM standard labels are recorded in EBCDIC. When reading a tape, the LABEL parameter can take a fourth option, ANY; in this case DMS sets the label field of the UFB after reading the beginning of the tape.

A non-labelled (NL) tape may be a tape containing no labels, such as one created by the BACKUP utility, or a tape containing labels that correspond to neither the ANSI or IBM standards. When such a tape is read, DMS treats the labels as if they were the first data block(s) of the files.

NOTE

The ALLOWNL parameter is no longer needed to insure that DMS can process a non-labelled tape. Existing programs that use this UFBGEN parameter do not need to be changed.

You can access a labelled tape file by its file name or its file sequence number. You access a file on an unlabelled tape volume by its file sequence number. The first file on a magnetic tape volume is file sequence number one (FSEQ=1).

The tape file label, containing the name of the file, its block and record length, and other particulars, is divided into as many as three parts called headers. Both Wang ANSI and IBM file labels contain the same information. The first file label header (HDR1) contains the file and volume names and sequence numbers, the user ID, system, and date on which the tape was created. The second header (HDR2) includes the type of file, the block length, and the length and type of records. You can use the TAPEDUMP useraid to print the exact contents of tape volume and file labels.

When writing a file to tape, you must select either partial or full tape file labels. The UFBGEN parameter HEADER=FULL is the default, formatting file labels with both HDR1 and HDR2. If HEADER=PARTIAL, DMS formats only the HDR1 as a file label. The third header, HDR3, is used in Wang OIS systems. A VS system can read a file containing a HDR3, but it can neither read nor write the HDR3.

UFBGEN statements for seven- and nine-track magnetic tape files are shown in Examples 12-1 and 12-2.

Example 12-1. A Sample UFBGEN Statement for Seven-Track Tape

```
ZOOTWO      UFBGEN     PRNAME=TWOZOO,FORG=CONSEC,DEVCLASS=MTAPE,
                       DEN=1600,LABEL=AL,HEADER=FULL,VLEN=YES,
                       BLKSIZE=8192,RECSIZE=2048,FSEQ=1,VSEQ=2,
                       EOD=EOV.
```

Example 12-2. A Sample UFBGEN Statement for Nine-Track Tape

```
ZOOFILE     UFBGEN     PRNAME=MYZOO,FORG=CONSEC,DEVCLASS=MTAPE,
                       TRACK7=YES,DEN=800,LABEL=NL,PARITY=ODD,
                       BLKSIZE=8192,RECSIZE=2048,FSEQ=1,VSEQ=2,
                       EOD=EOV.
```

12.3 TAPE BLOCKS, RECORDS, AND BUFFERS

DMS can process magnetic tape in RAM, BAM, or PAM access method. This means that data can be transferred by logical record (RAM), tape block units (BAM), or by an arbitrary sized physical unit (PAM). The latter two (BAM and PAM) are only accessible through Assembly language (see Chapter 10).

The unit most frequently used for BAM or PAM access is the tape block. A tape block is not necessarily 2K bytes in size; it can vary from 12 bytes to 32K (32,768) bytes. There is no Wang standard tape block size, although an even multiple of 2K is recommended for processing efficiency. Tape blocks are the only type of blocks in DMS that are not restricted to 2K bytes. You must specify the block size in bytes as the BLKSIZE parameter of UFBGEN.

12.3.1 Tape Records

Each tape block contains one or more records. In no case can a record be larger than a block or span tape blocks. The relationship between the size of a record and a tape block depends on the type of record.

Tape records can be fixed length, variable length using Wang's DMS record header data, variable length using the IBM variable length conventions, compressed according to Wang compression conventions, or undefined. An undefined record is one in which the record size defaults to the block size. Only one undefined record per block is permitted. DMS can compress data recorded in ASCII or EBCDIC using the Wang compression algorithm; DMS cannot uncompress data recorded using a non-Wang compression algorithm. Record size, variable record length, and compression are coded using the same UFBGEN parameters used for disk file definition: RECSIZE=n, VLEN=YES, COMP=YES.

For fixed length records, the tape block size should be an even multiple of the size of the records. For Wang variable length records, the block should be a multiple of the record size plus 4 bytes. For IBM-format variable length records, the block size should be a multiple of the record size plus 8. Block sizes can be computed using the following equations, in which n is the number of records per tape block:

Fixed Length	Blocksize = (recsize) x n
Wang Variable Length	Blocksize = (recsize + 4) x n
IBM Variable Length	Blocksize = (recsize + 8) x n

Thus, the maximum tape record size is 32,768 for fixed length records, 32,764 or 32,760 for variable length records. DMS can process blocks containing records larger than 2048 bytes using BAM or PAM. DMS cannot process records larger than 2048 bytes in RAM. When a record larger than 2048 is read from tape, DMS divides it into 2048-byte sections. These 2048-byte sections are recorded on disk as 2048-byte, fixed length records.

Unlike disk processing, tape records have a minimum record size. An extremely short record cannot be distinguished from noise by tape error checking routines. For this reason, minimum tape record sizes are established, as shown in Table 12-1.

Table 12-1. Minimum and Maximum Record Sizes for Magnetic Tape

	Minimum Recsize	Maximum Recsize
Fixed Length Records Input to DMS	12 bytes	32,768 bytes
Fixed Length Records Output by DMS	18 bytes	32,768 bytes
Variable Length Records		
Wang Format (input)	8 bytes	32,764 bytes
IBM Format (input)	4 bytes	32,760 bytes
Variable Length Records		
Wang Format (output)	14 bytes	32,764 bytes
IBM Format (output)	10 bytes	32,760 bytes

12.3.2 Tape Buffering

DMS tape support provides main memory buffering to allow the processing of up to 32K bytes of data at a time. This buffering is available in all three access methods: BAM, PAM, and RAM. DMS buffers tape data by allocating two buffers, each the size of a tape block. This automatic double-buffering can reserve up to 64K bytes of main memory for the processing of each tape file. Thus, processing a magnetic tape with a large block size can adversely affect the response time of other operations on the VS, especially if the VS has a small main memory size.

Double buffering speeds tape I/O processing by allowing one of the buffers to handle the physical I/O to the tape drive, while record blocking or unblocking is performed in the other buffer. At the conclusion of a block I/O the two buffer blocks alternate functions.

12.4 TAPE FILE MODES AND FUNCTION REQUESTS

Like other DMS files, you must open a tape file in a particular mode before accessing its records or blocks. After you issue the OPEN statement, you can use READ, WRITE, and START function requests to process the file. When file processing is completed, your program must issue a CLOSE instruction. Features unique to magnetic tape processing are available for the CLOSE instruction.

12.4.1 Tape File Modes

You can open a magnetic tape file in three modes: Input, Output, or Extend. Input mode allows you to sequentially read a tape file into main memory using the READ function request. Output mode is the mode of tape file creation, in which WRITE function requests sequentially create a new tape file. When you add a new file to tape volume, use the Output mode and provide the file sequence number. It is not necessary to specify NRECS (the estimated number of records to be written) when creating a tape file.

If an ASCII tape file already exists, you can add more records (or blocks) to the end of the file using Extend mode. Extend mode is not supported for IBM-format tape. Specifying the Extend mode moves the record pointer to the end of the file; the first WRITE function request begins writing at that point. If any other data is stored on the tape, Extend mode erases and overwrites it. Thus, you should make sure that a file to be extended is the last (highest file sequence number) file on the tape volume.

Only one file at a time can be open on a tape volume. This is an important consideration when placing multiple files on a tape volume.

Users cannot share tape files. Unlike disk files, only one user can read a tape file at a time. A second user cannot access a tape volume until the first closes the tape file and relinquishes the volume. When you logically mount a tape volume, you have the choice of "Shared" or "Exclusive" modes. The "Shared" option on the Mount Tape screen allows a user other than the person who mounted the tape to claim the tape volume for exclusive access by issuing an OPEN statement. When you open a file on the tape volume, all other users are barred from accessing that tape volume. When you close the tape file, another user may issue an OPEN statement, thus claiming exclusive access to the tape volume. Do not confuse this type of device sharing with the DMS Shared mode used for disk files.

12.4.2 Tape Function Requests

Magnetic tapes can be read and written sequentially. Existing records cannot be updated or deleted. Files cannot be accessed randomly. For example, in order to read the fifth record in a file, you must sequentially read the previous four records. The only mode offering file positioning is Extend mode, which provides file positioning for writing at the end of the file. To return to the beginning of the file, issue a CLOSE instruction and reopen the tape file.

Three function requests are supported for magnetic tape processing: READ, WRITE, and START. In Assembly language, a function request must include a UFB=ufbname clause, and may include a COND= (conditional execution) clause. These function requests and their modifiers are described as follows:

READ	The READ function request sequentially reads logical records (in RAM) or physical units (in BAM or PAM) from a magnetic tape into the tape buffer block and places the record in the user record area. The first READ issued reads the first record on the file; each successive READ reads the next record. READ NEXT is synonymous with READ.
READ NODATA	A READ NODATA function request is similar to a READ statement, except that the record read is not placed in the user record area. Instead, the record is retained in the tape buffer block, and the address of the record within that buffer is placed in Register 1.
WRITE	The WRITE function request sequentially writes logical records (in RAM) or physical units (in BAM or PAM) from the user record area to the magnetic tape. A WRITE function request erases and overwrites any previous material on the portion of tape written to. In Output mode a new file is written to tape by issuing successive WRITE function requests. In Extend mode, successive WRITE function requests add new records to the end of the tape file.
START WAIT	The START WAIT function request causes the system to wait for the completion of the previous I/O operation. You should issue a START WAIT after each READ or WRITE instruction in PAM. This waiting function is performed automatically in BAM and RAM. START WAIT is only available in Assembly language.

12.4.3 The CLOSE Instruction

After all processing has been performed on a tape file, the user program must issue a CLOSE instruction. Each file opened must be closed; a file can be opened and closed several times in different modes during a program run. In programming languages that do not contain an OPEN or CLOSE statement (Fortran 66 and RPG II), the Open is caused by the first access of the file, and the Close is effected by either the end of file, or the end of the program run. If your program omits a CLOSE statement, DMS closes the file at the termination of the program run.

A CLOSE statement must contain a UFB=filename clause. Unlike CLOSE instructions for disk and other types of files, the tape CLOSE statement can be modified. CLOSE modifiers govern the automatic rewinding and logical dismounting of a tape volume. The allowed CLOSE statements are as follows:

CLOSE [no modifier]	Closes a tape file, deletes the Open File Block (OFB) control block, and deallocates main memory buffers. If you opened the file in Output or Extend mode, CLOSE writes end-of-file trailers (EOF1 and EOF2) and a tape mark. In addition, the CLOSE instruction rewinds the tape volume. The rewind stops at the tape mark indicating beginning of the tape data.
CLOSE UNLOAD	Performs all of the functions of the unmodified CLOSE instruction, including rewind. When you specify a CLOSE UNLOAD, the rewind continues past the first tape mark and unloads the tape leader from the takeup reel. CLOSE UNLOAD also logically dismounts the tape volume.
CLOSE NOREWIND	Performs all the functions of the unmodified CLOSE statement, except that DMS does not rewind the tape. The tape remains positioned at the point of the last operation. This instruction is most commonly used for reading or writing multiple files on a tape volume.
CLOSE REEL	This instruction does not in fact close a tape file. Instead, you use this instruction with multivolume files to close a reel of tape and continue file processing on a second tape volume. See Section 12-6.

12.5 MULTIPLE FILES ON A TAPE VOLUME

A magnetic tape volume, or reel, can contain more than one file. Multiple files are identified by their file sequence number (FSEQ) specified in UFBGEN; the first file on the volume is FSEQ=1. FSEQ values can range from 1 to 9999. If the tape is labelled (AL or IL), you can also identify a file by its file and library name. If a discrepancy exists between the FSEQ and the file name, DMS displays a screen that requires you to change one or the other value before proceeding.

Processing of multiple files on a volume does not depend on the relative positions of those files. For example, if you close FSEQ=7 with a CLOSE NOREWIND, then issue an OPEN on FSEQ=5, DMS automatically causes the tape drive to back up two files. You can read a tape file repeatedly without rewinding the tape volume.

CAUTION

The Extend mode should be used with extreme caution on multiple file volumes. Extend overwrites whatever data follows the file being extended. Consequently, only the last file on a volume should be extended.

12.6 TAPE FILES SPANNING MULTIPLE VOLUMES

A unique feature of tape files is that they can span volumes. Each volume is assigned a volume sequence number (VSEQ), so that at the conclusion of processing of a volume the system prompts the operator to mount the next volume in the sequence. You must read multiple tape volumes in sequence, beginning with Volume Number 1.

If you have opened a tape file in Output or Extend modes, a CLOSE REEL instruction writes an end-of-volume label and rewinds, unloads, and logically dismounts the tape volume. The system then prompts the operator to mount a new tape (formatted using TAPEINIT), assigns to it the next VSEQ number, and continues writing records on the new volume.

If you have opened a tape file in Input mode, an end-of-volume trailer calls a CLOSE REEL subroutine that rewinds, unloads, and logically dismounts the current tape and prompts the operator to mount the next tape in the sequence.

When creating a multivolume file, you should normally include an EOD=EOV parameter in the UFBGEN for the file. If specified, this parameter instructs DMS to write an end-of-volume trailer to the output file when it reaches the end of the input file data, and then to close and rewind the newly created output file.

12.7 OPTIONAL USE OF TAPE STORAGE

You can decide to use tape or disk as the storage medium for a file when you run the program that accesses the file. Using the allow tape option, you can decide to place a file on either disk or tape without changing the program.

12.7.1 Programming the Allow Tape Option

To provide for optional use of tape storage, the Assembly language programmer writes a ALLOWTAPE=YES parameter in UFBGEN. The DEVCLASS parameter value should be DISK; disk storage is the default for the allow tape option. The UFBGEN should include all parameters necessary for both disk and tape. DMS ignores parameters unique to one type of device when processing the file using the other device type. The values of parameters used by both tape and disk must be acceptable values for both device types: the file organization must be consecutive, the blocksize must be 2048, and the record size must be within the maximum record size for consecutive disk files of that record type.

The file access mode and the function requests used for processing the file are limited to those supported for tape: READ and READ NODATA in Input mode, and WRITE in Output or Extend mode. The CLOSE statement can include the modifiers provided for tape files; these have no effect when running the file on disk.

You can use the allow tape option for several files opened by a program. However, remember that only one file can be open on a tape volume at a time.

12.7.2 Running a Program with the Allow Tape Option

Prior to running a program with the allow tape option, the storage device must be attached. If tape is to be used, you must physically mount the tape reel on the tape drive, and logically mount the tape from the operator's console. The parity and density set on the drive should agree with the values specified in the program.

If you have specified ALLOWTAPE=YES, you must select the device type each time you run the program. When the program is run, the system displays a screen that requests the file, library, and volume name, and the device type. Specify the appropriate volume name (and, if necessary, change the file and library names) and specify a device type of either DISK or TAPE. Type this information on the screen, then press the ENTER key. DMS reads or writes the file on the specified device.

CHAPTER 13
PRINTER, PROGRAM AND WP FILES

13.1 INTRODUCTION

Printer, program, and word processing (WP) files are DMS consecutive files that contain specially formatted information.

13.2 PRINTER FILES

A printer file is a consecutive file containing variable length records. You can write data to a printer using either the Record Access Method (RAM) or the Physical Access Method (PAM). PAM enables you to write a block of records to a printer.

13.2.1 Defining a Printer File

To establish a printer file, you specify PRINT=YES in UFBGEN; this sets the UFBFORGPRINT bit in the UFB. You can define the file name, library, and volume for a print file, or you can omit these parameters and accept the system defaults. If you use the system defaults, DMS assigns the print file to the default library and volume for the print files of the person running the program. If the system cannot place the print file in the default volume, it selects another volume that both contains sufficient space and permits print file spooling.

The default name of a print file is the first four characters of the name of the program that creates the file, followed by a system-generated four-digit number. This number is initialized when a user logs on, and is incremented each time that user creates a work file, temporary file, or print file.

You can output a printer file directly to a printer, or you can output it to a disk or tape drive. You specify the destination of a printer file by specifying a value for the DEVCLASS parameter of the UFB. DEVCLASS=PRT writes the file directly to a printer, if possible; DEVCLASS=DISK stores the printer file on disk.

Print Files Written Directly to a Printer

If you are writing a printer file directly to a printer, you specify DEVCLASS=PRT, and specify the device number of the desired printer by coding the DEVNO parameter in UFBGEN. The printer must be detached and thus available to be reserved by your task. Your print mode default must be O (Online) for direct printing; otherwise, the print file is spooled to disk.

A printer file written directly to the printer is an output-only file; the only valid file access mode is MODE=OUTPUT. You cannot use DMS function requests to read or update a file that has been sent to a printer.

If you specify DEVCLASS=PRT, DMS always generates a print file name, and ignores any user-supplied file name. The file name respecification screen is not displayed.

You can specify the FORM number for a printer file in UFBGEN. You specify a user-defined FORM number of 1 through 255 (with zero as a default) to inhibit the printing of a printer file if the wrong kind of paper is mounted on the printer. The system will not print a file unless the form number the operator specifies for the printer is the same as the form number coded in the file's UFB.

You can also specify a PRTCLASS with a value of A through Z for each printer file in UFBGEN. If you do not specify the FORM or PRTCLASS in the UFB, you can assign these values at runtime by means of a GETPARM screen.

Print Files Stored on Disk

If you specify DEVCLASS=DISK (the default), DMS always writes printer files to disk, regardless of the print mode default value. If you specify DEVCLASS=PRT, and your print mode default is set to H (Hold), K (Keep), or S (Spool), DMS writes your printer files to disk. You should specify the UFBGEN parameters PRINT=YES, VLEN=YES, and COMP=YES for printer files written to disk. Printer files on disk are always compressed.

You write printer files to disk in Output mode. You can open an existing print file on disk in Input mode, Extend mode, or I/O mode for shared processing.

13.2.2 Defining Printer File Records

A printer file can contain an unlimited number of variable length records. Each record represents one line of text. You establish the length of a line of text (the maximum record size) by specifying a value for the RECSIZE parameter of the User File Block before opening the file. The maximum record size is 134 bytes. If you attempt to write a record to a printer file that exceeds the RECSIZE value you specified, DMS rejects the record with a File Status '97' (invalid length).

Printer file records appear as shown in Figure 13-1.

Record Length Indicator (2 bytes)	Printer Control Field (2 bytes)	Printer File Data (0 to 132 bytes)

Figure 13-1. A Printer File Record

Each printer record contains a 2-byte printer control field. This field appears before the record data and is used to control the printer carriage. You can set the value of this field to establish the number of lines to advance the paper, or to specify the vertical tabbing channel to select from the printer's form tape. The first byte of the printer control field dictates the type of carriage control operation to perform. The available carriage control options are listed in Table 13-1.

Table 13-1. Printer Control Field Options

Byte 0, Carriage Control Operations

Type of Control	Carriage Control Occurs	Character Width	Printer Alarm	Hex Code
skip lines	before printing	regular	no alarm	00
skip lines	before printing	regular	alarm	10
skip lines	before printing	expanded	no alarm	20
skip lines	before printing	expanded	alarm	30
skip lines	after printing	regular	no alarm	40
skip lines	after printing	regular	alarm	50
skip lines	after printing	expanded	no alarm	60
skip lines	after printing	expanded	alarm	70
tabbing	before printing	regular	no alarm	80
tabbing	before printing	regular	alarm	90
tabbing	before printing	expanded	no alarm	A0
tabbing	before printing	expanded	alarm	B0
tabbing	after printing	regular	no alarm	C0
tabbing	after printing	regular	alarm	D0
tabbing	after printing	expanded	no alarm	E0
tabbing	after printing	expanded	alarm	F0

Byte 1, Carriage Skip Count

If the value of Byte 0 is 70 or less, place the number of lines to skip in this field.	01 to 7F
If the value of Byte 0 is greater than 70 place the number of the channel to skip to for tabbing in this field.	01 to 0C

If you set the first byte of the printer control field (as shown in Table 13-1) to a hexadecimal value from 00 to 70, the printer will skip lines when printing the record. You can specify the number of lines skipped by coding the number of lines in the second byte of the printer control field. You can specify any value from hexadecimal 00 (strikeover) to hexadecimal 7F (skip 127 lines).

If you set the first byte of the printer control field (as shown in Table 13-1) to a hexadecimal value from 80 to F0, the printer will change the channel it is using on the printer's forms tape. This forms tape channel governs the vertical tabbing used by the printer. You can specify the forms tape channel by coding the channel number in the second byte of the printer control field. You can specify any value from hexadecimal 01 (Channel 1) to hexadecimal 0C (Channel 12). Channel 1 is the top-of-form channel on most printers.

Expanded characters can only be specified for matrix printers. If you specify that a printer file record is to be printed in expanded format, as specified in the printer control field, a maximum of 68 characters can be printed on each line. If a printer file record contains more than 68 characters, the printer will truncate the line after 68 characters. DMS does not report an error message for truncated printer records.

13.2.3 Writing Records to a Printer File

You use the WRITE function request to write records into a printer file. DMS writes print file records sequentially. You cannot use any other function requests on printer files.

The buffering default for printer files is one 2K buffer block. You can use the large buffer strategy to improve printer file performance. The large buffer strategy is described in Chapter 9. Using the large buffer strategy does not improve file output to a printer, but it does speed the output of a printer file to a disk storage device.

For further information on printer files, refer to the VS Principles of Operation manual.

13.3 PROGRAM FILES

A program file is a consecutive file of 1024-byte fixed length records. It contains the object code produced as output from a compile, assemble, or link operation.

Generally, you create program files by compiling source code produced using the EDITOR, which automatically specifies the file parameter values. You can specify program library and volume defaults using the Set Usage Constants option of the Command Processor.

When reading a program file in Input mode, you should specify PROG=YES in UFBGEN. This checks the input file to make sure that it is a program file. If the file is not a program file, DMS rejects the Open with a file status error.

13.4 WORD PROCESSING FILES

A word processing file is known as a document. A document is a consecutive file containing fixed length 256-character records. WP file names consist of a four-digit number. WP documents are stored in system-defined libraries on the WP volume. WP libraries are named DOCMNTxx. The xx portion of the library name can contain one or two letters. A single letter indicates an uppercase WP library (e.g., DOCMNTG contains WP library 'G'); a double letter indicates a lowercase

WP library (e.g., DOCMNTGG contains WP library 'g'). WP glossaries are stored in library DOCMNT0. Defective WP documents are stored in library BADDOCxx. The xx portion of the library name contains one or two letters, representing the WP document library.

All word processing files contain a file preface that records the number of keystrokes in the document, the date of creation, date of last update, etc. To open a WP file in Output mode, you should specify PLOG=YES in UFBGEN to inform DMS that a file prologue will be present. DMS ignores PLOG in all other modes.

PART IV
Error Routines and Special Case Applications

CHAPTER 14
DMS ERROR PROCESSING ROUTINES

14.1 UFB ERROR MONITORING

DMS preserves a record of the most recently performed operation in the User File Block (UFB). It records the last function request performed, the error status returned by that function request, and the user-specified error routine address.

14.1.1 File Currently Open -- UFBF1

The UFBF1 field contains various indicator flags set by the Open operation. One of these flags indicates whether you currently have the file open. When you open a file, DMS sets the low-order bit of UFBF1 (i.e.: 0000 0001). When you close the file, DMS resets this bit to zero, and sets the X'08' bit (i.e.: 0000 1000) to indicate that the file is closed, but has previously been opened and is hence ineligible for opening in Output mode. UFBF1 is a read-only field; you cannot open or close a file by setting bits in UFBF1.

14.1.2 Last Function Request -- UFBLF and UFBLFMOD

DMS uses the one-byte UFBLF field to record the last function request attempted on the file. DMS supplies one of the following hexadecimal values to UFBLF:

00	OPEN statement
04	READ function request
08	WRITE function request
0C	REWRITE function request
10	DELETE function request
14	START function request
18	CLOSE statement

DMS uses the one-byte UFBLFMOD field to record the function request modifier for the last function request successfully performed.

14.2 FILE STATUS ERRORS -- UFBFS1 AND UFBFS2

The UFB contains two file status fields UFBFS1 and UFBFS2. DMS uses UFBFS1 to indicate the general type of file status, and UFBFS2 to indicate the specific type of error within each file status. Each of these fields is one byte in length; in this manual the values for these fields are expressed as character values. The two-character file status code specifies the value for both UFBFS1 and UFBFS2: the first character of the file status code contains the UFBFS1 value, the second character contains the UFBFS2 value. For example, a file status of '95' consists of a UFBFS1 value of '9' and a UFBFS2 value of '5'. A value of '0' in UFBFS1 indicates successful completion.

For interactive DMS processing, UFBFS2 stores the AID character. If UFBFS2 is blank the keyboard is unlocked; otherwise DMS has received a workstation response and locked the keyboard.

When you open a file, DMS sets UFBFS2 to '0'. When an error exit is taken, DMS sets UFBFS2 to record which Open Exit condition is taken.

14.2.1 Open Exit Processing

If you specify an EXIT as part of an OPEN statement, the system stores the EXIT's bit mask value in the high-order byte of the Open parameter word on the stack. The Open parameter word also contains the UFB address. If a failure occurs on that Open, DMS set UFBFS1 to '9', indicates the type of failure UFBFS2, and sets UFBPREVO.

If an Open Exit occurs, DMS treats the UFBFS2 byte as a bit mask, and sets the bit in UFBFS2 that corresponds to the type of failure. The bit values are listed in the description of the Open Exit in Chapter 6.

If the UFBFS2 field contains the value X'04', a possession conflict occurred during Open processing. If the UFBFS2 field contains the value X'01', a format error occurred during Open processing. In either case, DMS places further information on the specific type of error in UFBXCODE. UFBXCODE can take the following values:

Possession Conflicts

X'00'	= No further explanation
X'01'	= Device in use.
X'02'	= Device detached.
X'03'	= Volume exclusion.
X'04'	= File possession conflict
X'05'	= Paging file - system only.
X'06'	= Image file
X'07'	= Already open - this user.
X'08'	= Already open - this user.

Format Errors

X'10'	= Program requires 7 track tape while drive is 9 track or vice versa.
X'11'	= UFB FORG = print while FDR FORG not equal to print.
X'12'	= UFB FORG = PROG while FDR FORG not equal to PROG.
X'13'	= UFB FORG = CONSEC while FDR FORG not equal to CONSEC.
X'14'	= UFB FORG = WP while FDR FORG not equal to WP.
X'15'	= UFB FORG = INDEXED while FDR FORG not equal to INDEXED.
X'16'	= UFB FORG neither CONSEC nor INDEXED - error.

14.2.2 Error Exit Routine Addressing -- UFBERRAD and UFBEODAD

When a fatal error status occurs, DMS checks the UFBERRAD (error routine address) and UFBEODAD (UFB end of data address) fields for a user specified error address value. You can supply an error routine address to these fields either before opening the file, or before issuing a function request during file processing.

DMS takes the UFBEODAD address for a file status with a character value greater than '09' but less than '30'. It uses the UFBEODAD error exit address for conditions in which the program reaches the end of the data, attempts to write an indexed record with an invalid key, or requests a record that cannot be found.

DMS takes the UFBERRAD address for a file status with a character value greater than '30'.

- If UFBERRAD contains an error routine address, DMS takes that address for a file status of '30' or greater. The normal return address is stored in Register R0.

- If UFBEODAD contains an error routine address, DMS will take that address for end-of-data and invalid key conditions (file status '10' through '29'). The normal return address is stored in Register R0.

- IF UFBEODAD is zero, but UFBERRAD contains an error routine address, DMS will take the UFBERRAD address for all error status codes. The normal return address is stored in Register R0.

- If both UFBEODAD and UFBERRAD are zero, the system abnormally terminates your program on any file status of '10' or greater. DMS issues a cancel message, indicating the condition that caused DMS to cancel your program. Cancel messages are identified by a three-character error number. DMS stores the number identifying the most recent cancel message in UFBDMSGID.

14.2.3 Error Messages -- UFBF4NOMSG

You can select whether DMS should display an error message when taking an error exit. If you set UFBF4NOMSG to 1, DMS does not display error messages when the program takes the error exit address stored in UFBERRAD. If you did not provide an UFBERRAD address, DMS ignores UFBF4NOMSG, and displays all errors.

14.2.4 Fatal Errors

A File Status of '60' or greater indicates a fatal error. A fatal error causes DMS to issue a CANCEL SVC and to store the cancel message number in UFBDMSGID. The cancel message describes the situation and gives the UFB address, the address of the function request, and the parameter reference name (prname) of the file. You can specify an ERRAD address, or allow DMS to cancel your program following a fatal error.

DMS issues a fatal error for certain file status error conditions. If UFBERRAD is zero (for file status greater than or equal to '30') or if both UFBEODAD and UFBERRAD are zero (for file status less than '30'), then you have not supplied a return address for file status error conditions. In this case, DMS cancels your program. The CANCEL message includes the file status value and a description of the condition that caused the file status.

Each time that DMS uses certain UFB parameters, it checks the values of those parameters. If any of these UFB fields are inconsistent or invalid, DMS cancels your program. The CANCEL message indicates which field of the UFB had the invalid value. Errors of this type are generally caused by improperly modifying DMS fields in the UFB. Many of the UFB fields are not user-modifiable.

If a program check occurs during the execution of a function request, the system cancels your program. The most likely reason for a program check during DMS processing is an invalid address in UFBRECAREA or UFBKEYAREA. You can identify a program check in DMS by inspecting the Program Check Word (PCW). If the current operation was a function request, and the PCW indicates that the program check was in Segment 0 (the system segment), this indicates a DMS program check. Register R1 in the JSCI save area contains the UFB address. The PCW is described in the VS Principles of Operation manual.

CHAPTER 15
ADVANCED CONCEPTS

15.1 INTRODUCTION

Advanced concepts are DMS functions that you may seldom use, if ever, either because they are only used in unusual applications, or because they operate unobtrusively without you being aware of them. This chapter is divided into sections by the type of file: consecutive files, indexed files, and alternate indexed files. It also includes a description of multiple record types.

15.2 FOR CONSECUTIVE FILES

15.2.1 Anticipatory Buffer Priming

A program performing a sequential read of a file reads records from the buffer to the user record area by issuing successive READ NEXT function requests. When it reads the last record in the buffer, DMS must perform an I/O operation to bring in the next block(s) from the data file to reload the buffer. When a program reads the last record in the buffer using a READ NEXT (or READ with no modifier) function request, DMS assumes that the next command will also be a READ NEXT. Operating on this assumption, DMS reloads the buffer with the next data block(s) before the program issues a request for the first record in that block. This is called anticipatory buffer priming. Processing proceeds more rapidly using this technique because DMS can perform an I/O operation concurrently with the processing of the next program instruction cycle.

If the next program instruction is not a READ NEXT, DMS must load the buffer again with the requested data before processing the next instruction. This is a time-consuming operation. You should keep anticipatory buffer priming in mind when coding DMS function requests, and try to avoid mixing frequent READ NEXT commands with other function requests.

Anticipatory buffer priming occurs when the end of a main memory buffer is reached. You can set the size of this buffer to any multiple of 2K from 2K to 18K bytes.

15.2.2 Using START END with Shared Consecutive Files

START END truncates a file at the current record pointer. Normally, you should use START END in I/O mode, in which your program has exclusive access to the file. However, there may be circumstances when you would want to use START END while performing I/O in Shared mode. To do so, you must issue an explicit hold on the entire file (using the START HOLD function request), then position the current record pointer and issue the START END function request. Issuing a START END on a shared consecutive file affects other users sharing the file as described below.

If you perform a START END while another user is performing a START SKIP, the START SKIP may fail. If you use a START END to truncate the file at record n and another user does a START SKIP to a record with a higher relative record number than n, that person's START SKIP operation will fail with a File Status '10' (end of file). DMS does not issue an end of file status if a user's current record pointer is higher than n, nor does a START SKIP fail if it skips in the negative direction to a record that was not deleted by the START END operation.

If you perform a START END and then write new records to the file, other users' READ operations succeed, even if the START END has made their current record pointer out of sync with the file's current physical state. For example, if a user is positioned at Record 10, then a sequential READ succeeds if a Record 11 exists in the file, even if it has moved since the user's last operation.

15.3 FOR INDEXED FILES

15.3.1 Calculating the Number of Blocks in an Indexed File

The number of blocks in an indexed file is equal to the number of data blocks plus the number of primary key index blocks. An alternate indexed file also contains an AXD1 block and at least one alternate key index block per alternate key path. A DMS/TX file contains two additional file recovery blocks.

Data blocks:

You can calculate the number of data blocks as follows:

Fixed Length Records:

$$DB = \frac{NREC \times RL}{2043}$$

Variable Length Records:

$$DB = \frac{NREC \times [RL + 2]}{2043}$$

where NREC is the number of records in the file and RL is the length of the longest record.

Primary index blocks:

The number of primary index blocks can be calculated as follows:

$$LPXB = \frac{DB \times [PK + 3]}{2043}$$

$$HPXB = \frac{LPXB \times [PK + 3]}{2043}$$

where DB is the number of data blocks and PK is the length of the primary key field. LPXB is the number of primary index blocks on lowest level of the index block tree and HPXB is the number of primary index blocks at each successively higher level of the tree. Repeat the HPXB equation until HPXB=1, substituting the PXB results from the previous HPXB equation. Add LPXB and the HPXBs to get the total number of primary index blocks.

15.3.2 Embedded Unused Blocks

When creating an indexed file, DMS attempts to place the index blocks in the exact middle of the assigned extent. This can result in unusable blocks embedded in the file if you greatly overestimate the number of records to be processed in Output mode. For example, if the estimated number of records is 1000, and the actual number of records written is 100, the index blocks will begin where DMS expected the 500th record to be, and the blocks between record 100 and the first index block are unused. These embedded unused blocks are chained together, and are used in preference to unused blocks at the end of the file for block splits. However, these blocks cannot be made available to other files by performing a release operation.

Even if you estimate the number of records accurately, embedded blocks between the last data block and the first index block can occur if the file records are highly compressed. DMS estimates the location of the center of the file based on uncompressed record lengths.

If a significant number of unused blocks are embedded in a file, you can remove them by running the COPY utility with REORG, and lowering the estimated number of records in the file. In the case of highly compressed files, it may be necessary to significantly underestimate the number of file records.

15.3.3 Extension Rights

Extension rights are a feature of DMS that enables one user at a time to incrementally claim and hold multiple resources in indexed files. The file sharing provided by extension rights is less comprehensive than that provided by the VS DMS/TX facility. You should write new application programs using DMS/TX multiple resource sharing, rather than extension rights. The use of extension rights does not affect DMS/TX sharing. DMS/TX is described in the VS DMS/TX Reference

When you claim extension rights, you are granted the ability to hold an unlimited number of indexed files, records, or generic key groups on a claim-as-needed basis. Only one user on the system can claim extension rights. You must release extension rights to enable another user to claim them.

Extension rights are supported in COBOL, RPG II, and Assembly language. In Assembly language, you issue a GETXRTS macroinstruction to claim extension rights. You must not be explicitly holding any resources when you request extension rights; if you are implicitly holding a record, it is automatically released when you claim extension rights. If another user has already claimed extension rights, your program waits for the period specified in the timeout option, then, if extension right remain unavailable, GETXRTS fails with a File Status '95'.

The interactions between extension rights and DMS/TX files and users are described in the VS DMS/TX Reference. You cannot perform a Hold List operation while holding extension rights. If you issue a Hold List while holding extension rights, DMS claims each item in the list as requested, rather than holding all the items on the list at once.

You issue a FREEXRTS macroinstruction to release extension rights. You do not have to release held resources before releasing extension rights. However, once you have released extension rights you cannot claim a new resource until you release all previously held resources. The START RELEASE function request does not release extension rights. However, the system does release extension rights at program completion.

15.4 FOR ALTERNATE INDEXED FILES

15.4.1 Overlapping Primary and Alternate Keys

It is possible to overlap the key fields of the primary and alternate keys, or even to completely embed one key within the other. An example of this is using a full name for a unique primary key, and using just the last name for an alternate key that allows duplication.

Key fields in data records are never compressed, so key overlap should have no significant effect on data record length. Alternate index pseudo-records (paired alternate and primary key fields) are not compressed, but they are concatenated so that character strings that appear in the primary key are not duplicated in the corresponding alternate key pseudo-record field. This can result in substantial space savings (up to 50%) in alternate index tree space allocation.

15.4.2 Extremely Long Alternate Keys

Avoid long alternate keys. When designating the alternate key field, keep in mind the following:

- The alternate key does not have to include an entire data field. For example, the alternate key does not have to include a person's entire last name, although the data record does.

- You can create code characters in the file to distinguish otherwise duplicate alternate key values. For example, a user program might print out "Johnson, Albert P.", from a record that says: "AJohnson Albert P.", in which "AJohn" is the alternate key.

- Alternate key values do not have to be unique. Unique alternate keys are preferable, as they do not require duplication of the primary key at the upper levels of the alternate key tree. However, if the alternate key can be reduced by several bytes, and the primary key is short, allowing duplication is often preferable to creating long unique alternate key values.

A long alternate key has a disproportionate effect on extent allocation, because DMS allocates the work area for alternate keys based on the file's longest alternate key field. This may result in a primary allocation larger than the actual needs of the file. The COPY utility also uses the longest alternate key to estimate space requirements when reorganizing the file. It is possible for a reorganized file of this sort to be longer than the original file. It is even possible to build (by adding records in I/O mode) an extremely large file that you cannot reorganize on that volume.

15.4.3 How an Alternate Index Tree is Built

When creating an alternate indexed file, you estimate the number of records to be put initially into the file, and supply the number, length, and location of alternate keys. DMS calculates a primary extent length by determining the number of data blocks, based on the maximum uncompressed length of the estimated number of records, plus 2 bytes per record for the alternate index mask. DMS then adds the estimated number of primary index blocks, based on the length of the primary key, and the number of low-level alternate index blocks based on the primary and alternate key lengths and on the assumption that one-half of the records will be on each alternate index path.

If an extent of that length is unobtainable, DMS assigns the largest available free extent of less than the calculated length. If initial file creation requires more than one extent, DMS automatically allocates up to two additional extents (each one-half the length of the primary extent) when the last block of the primary extent is filled. Because DMS calculates the block allocation for only the low-level blocks of alternate index trees, and estimates 50% of the records on each path, it often uses more than one extent in alternate indexed file creation. The additional extent generally contains alternate index tree records. If file creation fills all three extents, the Output mode WRITE cancels with a boundary violation. You must then estimate a larger number of records and restart the program.

DMS sets aside relative block 0 of the file for the AXD1 and begins placing data records in blocks, starting with block 1. You must present the records to DMS in ascending ASCII primary key sequence.

As each block of records is written, DMS copies the primary key of the final record in the block to a low-order index block that it established in the middle of the file. DMS creates the low-order primary index blocks at the same time as the data blocks, in primary key sequence. DMS creates the high-order index blocks upon completion of record writing in Output mode, and it places high-value FFFs along the right-hand edge of each level of the index block tree.

While DMS formats low-order primary index blocks, it also collects primary and alternate key record pairs (pseudo-records) in a work area at the high end of the first extent. The record pairs are at this point in primary key sequence. The layout of an alternate indexed file at this point is shown in Figure 15-1.

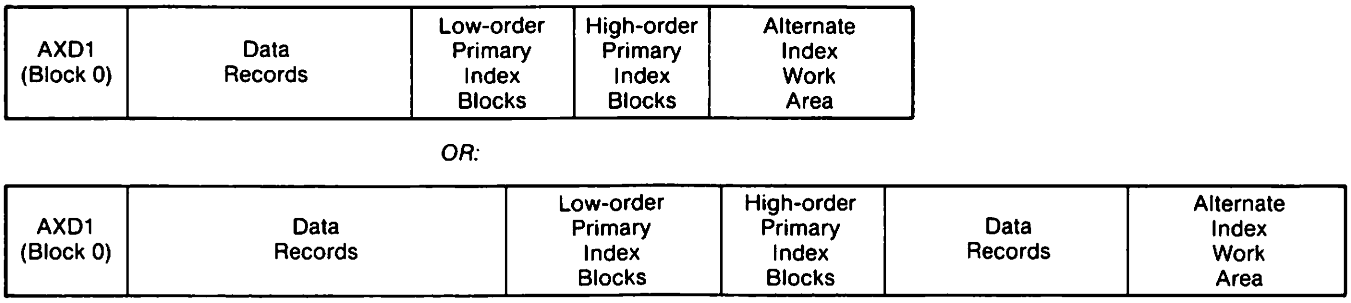

AXD1 (Block 0)
Data Records
Low-order Primary Index Blocks
High-order Primary Index Blocks
Alternate Index Work Area
OR:
AXD1 (Block 0)
Data Records
Low-order Primary Index Blocks
High-order Primary Index Blocks
Data Records
Alternate Index Work Area

Figure 15-1. Alternate Indexed File at WRITE Completion

When your program closes the file, DMS calls the BUILDALT routine, which constructs the alternate index trees. BUILDALT copies the pseudo-records in the work area to a temporary work file, sorts them into ascending ASCII sequence by alternate key, and then rewrites them into the work area, overwriting the original pseudo-record pairs. How BUILDALT uses file space for this operation is shown in Figure 15-2.

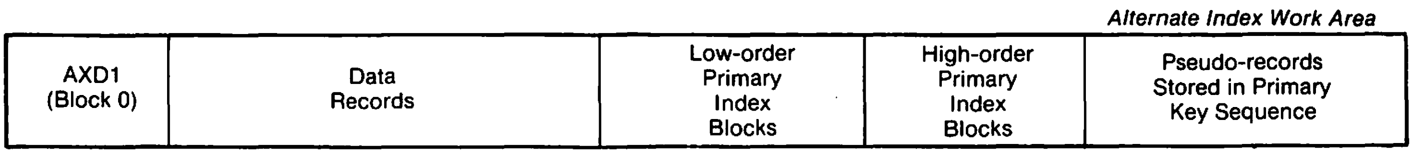

Alternate Index Work Area
AXD1 (Block 0)
Data Records
Low-order Primary Index Blocks
High-order Primary Index Blocks
Pseudo-records Stored in Primary Key Sequence

Figure 15-2. Alternate Indexed File during BUILDALT Processing

After BUILDALT creates the low-order pseudo-record blocks, it generates the high-order blocks of the alternate trees, and stores the relative block address of the root block of each tree in the AXD1. This completes alternate indexed file creation. The final file layout is shown in Figure 15-3.

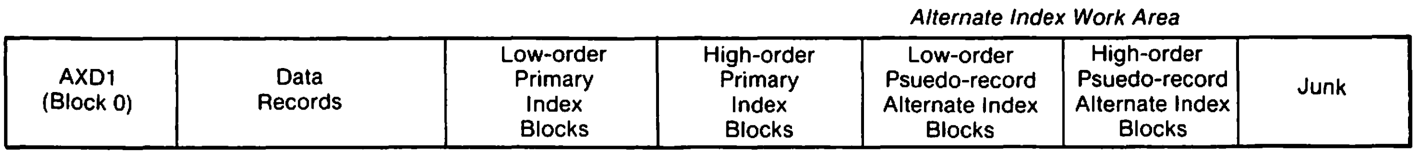

Alternate Index Work Area
AXD1 (Block 0)
Data Records
Low-order Primary Index Blocks
High-order Primary Index Blocks
Low-order Psuedo-record Alternate Index Blocks
High-order Psuedo-record Alternate Index Blocks
Junk

Figure 15-3. Alternate Indexed File at BUILDALT Completion

Usually, sorting the alternate index pseudo-records requires more work area space than the final alternate tree structure and occasionally DMS must allocate an additional extent. At the completion of BUILDALT processing, this extra space appears as garbage at the end of the file. You may decide to ignore these blocks, assuming that they will eventually be used for block splitting and file extension. Or you may free these blocks by running the COPY utility on the file with REORG=NO and RELEASE=YES. This process is considerably faster than running COPY with REORG=YES.

15.4.4 Creating an AXD1 in Segment 2

You can create an AXD1 area in the buffer (heap) storage area, by assigning buffer space. However, make sure that the space allocated is sufficient for the number of alternate key fields on the AXD1. Failure to provide a space large enough results in DMS assigning another area in the buffer for the AXD1; this area is functional, being addressable by the DMS, and is locatable via UFBALTPTR. Attempting to address a too-large AXD1 in the buffer can result in your processing alternate indexed records against non-existent (garbage) AXD1 fields. This usually causes the rejection of valid alternate indexed records, due to the failure of their MASK area to match the garbage data in the AXD1 PMASK area.

15.5 MULTIPLE RECORD TYPES

15.5.1 File Design Aspects

The VS supports COBOL coding for multiple record types. Multiple record type support is a programming method; it is not a structural feature of DMS. The use of multiple record types allows a program to define the fields differently for different records in a file. It does not affect the formatting of records performed by DMS.

For example, in a zookeeper's medical history file, records for male and female animals will contain different fields. A record for a female animal will contain a 10-byte maternity field in addition to the 100-bytes of medical history common to both males and females. The zookeeper can create this file with two different types of records by making all records in the file variable-length, with a maximum record length of 110-bytes. To access the two record types, the zookeeper codes two separate file descriptors in the Data Division of the program accessing the file, as shown in Figure 15-4.

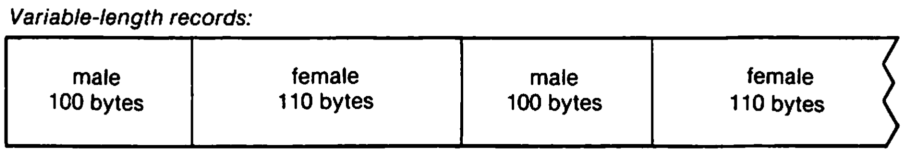


Figure 15-4. Multiple Record Types in a Data File

While the use of multiple record types is primarily a COBOL coding problem, you should decide on the method of locating the desired subset of the file at the time the file is created. There are three methods of arranging a file's records to facilitate multiple record type processing: sequencing, flagging, and alternate keying.

You can sequence the records in a file, so that the program can determine which file descriptor to apply based on a counter written into the program. For instance, in Figure 15-4, all even-numbered records are matched to one file descriptor, and all odd-numbered records to another. This method is restricted to consecutive files, preferably stable files, because the addition or deletion of a single record can throw off the counter.

Flagged records contain a field that indicates which record type they belong to. The program checks this flag and applies the appropriate file descriptor. In a consecutive file, this flag can be any byte(s). In an indexed file, you can establish the flag byte as part of the primary key. DMS sorts records added to a file in I/O mode according to their primary key values. If the first byte of the primary key is the flag byte, DMS will process all records of the first file type before any records of the second file type.

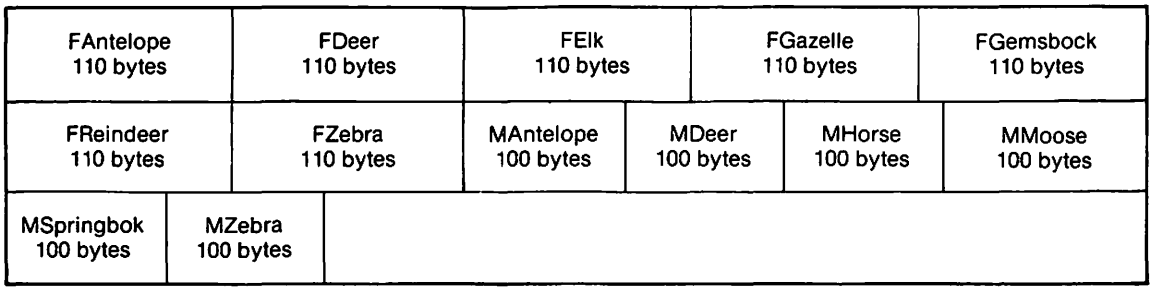


Figure 15-5. A Block of Records Showing Multiple Record Code Characters

A third method of creating files for multiple record access is by creating an alternate index key for each type of record. Only records of the same type would be placed on that type's alternate key path. This method is especially useful if only a few records are of a particular record type.

For example, you could establish an alternate index for all zoo animals that are currently on loan from other zoos. These longer records include the name and address of the owner, the date of loan, reason for loan, and anticipated date of return. Only a small percentage of zoo animal records are on this alternate key path. Therefore, all records on this alternate key path can be located more rapidly by using an alternate key than by sequentially reading every record in the file to check a flag field.

15.5.2 Records Larger than One Block

One use of multiple record types is to permit records larger than one block in length. If you use the multiple record flag as the last byte(s) of the primary key, DMS always sequentially processes this group of records in the same order. You can hold the entire group by issuing a hold by generic key value. Since your program uses the flag byte to select the file descriptor, the fields in the first record and the fields in the record following it can be completely different. In effect the two records are Part 1 and Part 2 that together make up a larger record. An example of multiple record pairs is shown in Figure 15-6.

Variable Length Records

PK: Antelope1 500 bytes	PK: Antelope2 150 bytes
PK: Buffalo1 500 bytes	PK: Buffalo2 150 bytes
PK: Caribou1 500 bytes	PK: Caribou2 150 bytes

Figure 15-6. A Block of Records Showing Multiple Record Pairs

PART V
Appendices

APPENDIX A
DMS CONTROL BLOCKS

A.1 THE UFB DSECT

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

*        THE USER FILE BLOCK (UFB) IS SUPPLIED IN THE USER'S
*        MODIFIABLE AREA BY THE USER'S PROGRAM BEFORE OPENING
*        A FILE, AND IS ADDRESSED TO REQUEST EACH OPERATION
*        ON THAT FILE.  THE ADDRESS OF THIS BLOCK IS PLACED
*        IN THE OPEN FILE BLOCK BY 'OPEN', AND THE ADDRESS OF
*        THE OPEN FILE BLOCK IS PLACED IN THIS BLOCK.
*
*        DATE  03-18-83
*        VERSION 6.00.23
*******************************************************************

UFBBEGIN          DS    OF            (FULLWORD ALIGNMENT REQUIRED

*******************************************************************
* ACCESS METHOD SECTION
* NO FIELDS NEED BE SUPPLIED BEFORE 'OPEN', BUT UFBERRAD
* UFBEODAD, UFBRECAREA, AND UFBKEYAREA MAY BE PRESET
* IF DESIRED. AFTER 'OPEN', THE USER'S PROGRAM NORMALLY
* HAS OCCASION TO MODIFY ONLY THIS SECTION OF THE UFB.
* THE FIRST BYTES OF EACH OF UFBVREAD, UFBVWRITE, UFBVREWRITE,
* UFBVDELETE AND UFBVSTART ARE ZEROED BY 'OPEN' AND SET
* THEREAFTER TO FUNCTION MODIFIER VALUES BY THE USER'S PROGRAM.
* THE SUCCEEDING BYTES OF THESE FIELDS CONTAIN ADDRESSES
* SUPPLIED BY 'OPEN' WHICH SHOULD NOT BE ALTERED BY THE
* USER'S PROGRAM WHILE THE FILE IS OPEN.
* UFBFS1 AND UFBFS2 ARE SET TO X'30' BY 'OPEN' AND MODIFIED
* THEREAFTER BY DATA MANAGEMENT FUNCTIONS.
*************************************************************

UFBVECT           DS    5A            BRANCH POINTS TO ACCESS
*                                     METHOD ROUTINES
*************************************************************
```

```
* THE FOLLOWING FUNCTION MODIFIER VALUES ARE PLACED IN THE FIRST
* BYTE OF THE WORD CONTAINING THE ADDRESS OF THE FUNCTION TO BE
* PERFORMED FOR A USER PROGRAM BEFORE BRANCHING TO THE ROUTINE
* ADDRESS.
               ORG UFBVECT
UFBV            DS    0F           (PREFIX TO EQUATE LABELS)
* MODIFIERS FOR READ:
UFBVHOLD         EQU   X'01'        (HOLD BLOCK EXCLUSIVELY)
UFBVREL          EQU   X'04'        (RELATIVE READ)
UFBVKEYED        EQU   X'04'        (KEYED READ)
UFBVNODATA       EQU   X'08'        (DO NOT MOVE DATA TO WORK
*                                    AREA ON READ)
* MODIFIER FOR WRITE/DELETE (RELATIVE DISK ONLY)        25

UFBVEOF          EQU   X'02'        (WRITE OR DELETE EOF)
25
* MODIFIERS FOR READ OR REWRITE (WORKSTATION ONLY):
UFBVTABS         EQU   X'10'        (READ OR REWRITE TABS - WS)

* MODIFIERS FOR READ (WORKSTATION ONLY):
UFBVMOD          EQU   X'02'        (READ MODIFIABLE - WS)
UFBVALTR         EQU   X'40'        (READ ALTERED - WS)
* MODIFIERS FOR REWRITE (WORDSTATION ONLY):
UFBVSELW         EQU   X'40'        (REWRITE SELECTED - WS)
* MODIFIERS FOR START (INPUT, IO, SHARED MODES; INDEXED DISK ONLY):
UFBVEQ           EQU   X'01'        (EQUAL TO)
UFBVGT           EQU   X'02'        (GREATER THAN)
UFBVGE           EQU   X'03'        (GREATER THAN OR EQUAL TO)
* MODIFIERS FOR START (INPUT, IO, SHARED MODES; RELATIVE DISK ONLY):25
UFBVLT           EQU   X'10'        (LESS THAN)      25
UFBVLE           EQU   X'11'        (LESS THAN OR EQUAL TO)
25
* MODIFIER FOR START (SHARED MODE; IGNORED FOR INPUT & IO MODES):

UFBVHFILE        EQU   X'80'        (HOLD FILE)
UFBVRLS          EQU   X'20'        (RELEASE HELD FILE)
UFBVRANGE        EQU   X'04'        (HOLD REQUEST FOR A RANGE)
UFBVRETRIEVAL    EQU   X'40'        (HOLD CLASS IS RETRIEVAL)
UFBVLIST         EQU   X'10'        (LIST OPTION)
* MODIFIERS FOR START (CONSECUTIVE OUTPUT & EXTEND MODES ONLY):
UFBVINPUT        EQU   X'04'        (CHANGE TO TEMPORARY IO MODE
UFBVOUTPUT       EQU   X'08'        (CHANGE TO OUTPUT MODE)
UFBVEXTEND       EQU   X'20'        (CHANGE TO EXTEND MODE)
* MODIFIERS FOR START (CONSECUTIVE FILES, INPUT AND I/O MODES ONLY):
UFBVBEGIN        EQU   X'10'        (BEGINNING OF FILE)
UFBVSKIP         EQU   X'40'        (FROM CURRENT RECORD
*                                    USING SIGNED WORD
*                                    ADDRESSED BY KEYAREA)
* MODIFIER FOR START (CONSEC FILES IN RAM I/O MODE):
UFBVEND          EQU   X'02'        RESET END OF FILE
* MODIFIERS FOR START (PHYSICAL ACCESS METHOD ONLY):
UFBVCMD          EQU   X'80'        (***VAGUE NOTE***)
UFBVWAIT         EQU   X'40'        (WAIT FOR I/O COMPLETION)
```

```
UFBVWAITS          EQU   X'41'          WAIT FOR TC I/O COMPLETION
*                                        ON THIS DEVICE ONLY
UFBVWAITM          EQU   X'42'          WAIT FOR TC I/O COMPLETION
*                                        ON ALL DEVICES OPENED BY
*                                        THIS PROGRAM
UFBVWAITA          EQU   X'43'          WAIT FOR TC I/O COMPLETIONS
*                                        AND TC UNSOLICIT INTERRUPTS
UFBVHALTIO         EQU   X'20'          HALT TC IO OPERATION
* MODIFIERS FOR START (WORKSTATION ONLY):
UFBVATTNT          EQU   X'10'          (TEST FOR ATTENTIONS RECEIVE
* *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
                    EJECT
                    ORG UFBVECT
UFBVREAD           DS    A              ..FOR READ
UFBVWRITE          DS    A              ..FOR WRITE
UFBVREWRITE        DS    A              ..FOR REWRITE
UFBVDELETE         DS    A              ..FOR DELETE
UFBVSTART          DS    A              ..FOR START
                    SPACE
                            ORG UFBVWRITE
UFBFLAGSD          DS        BL1 RUNTIME FLAGS FOR DISK PROCESSING
UFBFLAGSDNEWAXD    EQU   X'80'          ALT-INX FILE IS NEW FORMAT
UFBFLAGSDCONVPR    EQU   X'40'          AK/PK CONVERTED TO PSEUDO-REC
UFBFLAGSDMULTIO    EQU   X'20'          PSB WRITTEN WITH MULTIO FLAG ON
UFBFLAGSDXCASE     EQU   X'10'          3-WAY BLOCK SPLIT INDICATOR
UFBFLAGSDIODONE    EQU   X'08'          LAST ALTERNATE INDEX PROCESSED
*                                        (USED ONLY IF DFLAGSMULTIO ON)
                   DS     AL3           RESET ASSEMBLY COUNTER
                   DS     3A             . . .
                    SPACE
* THE FOLLOWING FOUR FIELDS MAY BE SET BEFORE 'OPEN' OR
* BEFORE THE FIRST FUNCTION AFTER 'OPEN'. THEY MAY BE CHANGED
* BY THE USER'S PROGRAM BEFORE ANY FUNCTION. IF UFBEODAD IS 0,
* UFBERRAD WILL BE USED FOR END OF DATA AND INVALID-KEY CONDITIONS.
* IF UFBERRAD IS 0, ABNORMAL TERMINATION WILL OCCUR ON ANY
* ERROR (AND ON THE ABOVE CONDITIONS IF UFBEODAD IS 0 ALSO).
UFBERRAD           DS    A              I/O UNUSUAL CONDITION USER
*                                       ROUTINEENTRY POINT, OR ZERO
UFBEODAD           DS    A              END OF DATA AND INVALID KEY
*                                       USER ROUTINE
*                                       ENTRY POINT, OR ZERO.
UFBRECAREA         DS    A              ADDRESS IN USER-MODIFIABLE S
*                                       OF RECORD WORK AREA
UFBKEYAREA         DS    A              ADDRESS OF AREA CONTAINING
*                                       SUPPLIED KEY OR RECORD NUMBER
*                                       FOR START OR READ FUNCTIONS
*                                       (IF ZERO FOR WORKSTATION FILES,
*                                       LINE NUMBER (ROW) TAKEN FROM ORDER
*                                       AREA)
UFBFS1             DS    CL1            FILE STATUS BYTE 1 FOR DMS
UFBFS1SUCCESS      EQU   X'30'          SUCCESSFUL COMPLETION
UFBFS1ATEND        EQU   X'31'          AT END
UFBFS1INVKEY       EQU   X'32'          INVALID KEY OR RECORD NO.
UFBFS1IOERR        EQU   X'33'          PERMANENT I/O ERROR
```

```
.** UFBFS1ADMSERR         equ x'34'     ADMS FUNCTION ERROR
UFBFS1CANCEL       EQU   X'36'          CANCEL CODE STORED
*                        FOR UFBF1NOMSG (OPEN,DMS,CLOSE); UFBFS2=C'0'
*                        MSGID AT UFBVREAD FOR O/C; NO MSGID IF DMS
UFBFS1TIME         EQU   X'37'          TIME-OUT CONDITION ON
*                                       SHARED MODE RESOURCE WAIT
UFBFS1SHARE        EQU   X'38'          FS FOR SHARER CONDITION
*                                       RESOURCE WAIT
UFBFS1OTHER        EQU   X'39'          OTHER CONDITIONS
**
UFBFS2             DS    CL1            FILE STATUS BYTE 2 FOR DMS
*
UFBFS2NOINFO       EQU   X'30'          NO FURTHER INFO
**
* THE FOLLOWING UFBFS2 VALUES ARE SET WITH UFBFS1INVKEY (X'32')
**
UFBFS2SEQERR       EQU   X'31'          SEQUENCE ERROR
UFBFS2DUPKEY       EQU   X'32'          DUPLICATE KEY
UFBFS2NOREC        EQU   X'33'          NO RECORD FOUND
UFBFS2BYVIOL       EQU   X'34'          BOUNDARY VIOLATION
**
* UFBFS2BDYVIOL IS ALSO USED WITH UFBFS1IOERR (FS = C'34')
**
.*** UFBFS2 values corresponding to FBFS1ADMSERR were defined here
* THE FOLLOWING UFBFS2 VALUES ARE SET WITH UFBS1SHARE (X'38')
**
UFBFS2ACC          EQU   X'35'          UPDATE ACCESS DENIED FOR
*                                       USER WITH READ-ONLY RIGHTS
*                                       IN SHARED MODE
UFBFS2RESERR       EQU   X'36'          RESOURCE CONTROL ERROR
UFBFS2DEADLOCK     EQU   X'37'          DEADLOCK
**
* THE FOLLOWING UFBFS2 VALUES ARE SET WITH UFBFS1OTHER (X'39')
**
UFBFS2ROLLBK       EQU   X'33'          CURRENCY LOST DURING
*                                       ROLLBACK
UFBFS2INVFUN       EQU   X'35'          INVALID FUNCTION OR
*                                       FUNCTION SEQUENCE
UFBFS2INVCMD       EQU   X'36'          INVALID COMMAND (ALIGNMENT
*                                       OR ADDRESS ERROR FOR DIRECT 1/O)
UFBFS2INVLTH       EQU   X'37'          INVALID LENGTH
UFBFS2MASK         EQU   X'38'          INVALID ACCESS MASK
*                                       (ALTERNATE INDEXED FILES)
UFBFS2TRLERR       EQU   X'38'          TRAILER COUNT NOT EQUAL
*                                       TO BLOCKS READ (SET BY SVC
*                                       CLOSE ONLY)
UFBFS2FMTERR       EQU   X'39'          FORMAT ERROR (BLOCK PREFIX,
*                                       RECORD PREFIX,EXPANSION ERROR OR
*                                       INVALID CHAIN FIELD)
**
* NOTE: UFBFS2 CONTAINS THE TERMINATING ATTENTION CHARACTER (AID BYTE)
*       ON WORKSTATION READ SUCCESSFUL COMPLETION.
```

```
**
* NOTE: THE FOLLOWING UFBFS2 VALUES ARE SET ONLY IF AN SVC OPEN
*       EXIT IS TAKEN. THESE VALUES ARE ALSO USED WHEN CREATING
*       THE OPEN EXIT MASK TO BE SUPPLIED TO THE OPEN SVC.
UFBFS2XFILE       EQU   X'80'          DUPLICATE FILE OR
*                                      FILE NOT FOUND
UFBFS2XLIB        EQU   X'40'          LIBRARY NOT FOUND
UFBFS2XVOL        EQU   X'20'          VOLUME NOT MOUNTED
UFBFS2XSPACE      EQU   X'10'          NO SPACE ON VOLUME
UFBFS2XVTOC       EQU   X'08'          NO VTOC SPACE ON VOLUME
UFBFS2XTAPELD     EQU   X'08'          WRONG TAPE LABEL/DENSITY
UFBFS2XPOS        EQU   X'04'          POSSESSION CONFLICT
UFBFS2XPROT       EQU   X'02'          PROTECTION CLASS VIOLATION
UFBFS2XFORMAT     EQU   X'01'          OPEN FORMAT ERROR - ERROR
*                                      CLASS DESCRIBED IN UFBXCODE
UFBAMEND          EQU   *
UFBAMLENGTH       EQU   (UFBAMEND-UFBBEGIN)
                   EJECT
* *************************************************************
* FILE LOCATION AND ATTRIBUTE SECTION
* ALL FIELDS IN THIS SECTION MUST BE SET (SOME OF THEM POSSIBLY
* TO 'NULL' VALUES) BY THE USER'S PROGRAM BEFORE INITIALLY
* ADDRESSING AN 'OPEN' TO THE UFB.
* ALL RELEVANT FIELDS AND FLAGS SET NULL BEFORE 'OPEN' ARE SUPPLIED
* HERE BY 'OPEN' PROCESSING AND MAY BE EXAMINED BY THE USER'S
* PROGRAM. THE PROGRAM SHOULD NOT MODIFY THESE FIELDS BETWEEN
* 'CLOSE' AND A SUCCESSIVE 'OPEN' IF THE SAME FILE IS REQUIRED
* (WITHOUT REPROMPTING).
* *************************************************************
UFBBLKSIZE        DS    H              MAGNETIC TAPE - MUST CONTAIN
*                                      PHYSICAL BLOCK SIZE BEFORE OPEN
*                                      IF OUTPUT MODE OR UNLABELLED
*                                      TAPE.
*                                      DISK OR DISKETTE - ALWAYS 2048
*                                      AFTER OPEN EXCEPT WHEN USING
*                                      PHYSICAL ACCESS METHOD (PAM)
UFBRECSIZE        DS    H              LOGICAL RECORD SIZE
*                                      (MUST BE SUPPLIED BEFORE OPEN FOR
*                                      OUTPUT OPEN MODE)
*
UFBFORG           DS    BL1            FILE ORGANIZATION
UFBFORGCONSEC     EQU   X'01'          CONSECUTIVE
UFBFORGINDEXED    EQU   X'02'          INDEXED
UFBFORGWP         EQU   X'04'          WORD PROCESSING FILE
UFBFORGVIBM       EQU   X'08'          IBM VARIABLE-LENGTH RECORDS
UFBFORGREL        EQU   X'08'          RELATIVE
UFBFORGU          EQU   X'10'          UNDEFINED RECORD FORMAT
UFBFORGVLEN       EQU   X'20'          VARIABLE-LENGTH RECORDS
UFBFORGPRINT      EQU   X'40'          PRINT FILE
UFBFORGPROG       EQU   X'80'          PROGRAM FILE
```

```
*
UFBF1             DS    BL1              OPTION FLAGS
UFBF1NOGET        EQU   X'80'            USE GETPARM = TYPE RD
UFBF1NODISP       EQU   X'40'            USE GETPARM = TYPE ID
* UFBF1NOGET AMD UFBF1NODISP USED BY SVC OPEN ONLY; NOT RESET BY DMS
UFBF1PAM          EQU   X'20'            PHYSICAL ACCESS METHOD
UFBF1BAM          EQU   X'10'            BLOCK ACCESS METHOD
UFBF1PREVO        EQU   X'08'            THIS UFB PREVIOUSLY OPENED
UFBF1WORK         EQU   X'04'            SCRATCH THIS WORK FILE ON
*                                        CLOSE IF SET & FILE HAS A
*                                        TEMPORARY NAME
UFBF1POOL         EQU   X'02'            BUFFER POOLING FOR RAM
*                                        (UFBBUFSTART MUST CONTAIN
*                                        BCT ADDRESS AT OPEN TIME)
UFBF1OPEN         EQU   X'01'            THIS UFB OPEN IF SET
UFBF2             DS    BL1              OPEN MODE FLAGS
.** UFBF2ADMS              equ x'80'         open in ADMS mode
UFBF2OUT          EQU   X'40'            TO OPEN FOR OUTPUT MODE
UFBF2IN           EQU   X'20'            TO OPEN FOR INPUT MODE
UFBF2IO           EQU   X'10'            TO OPEN FOR IO MODE
UFBF2EXTEND       EQU   X'08'            TO OPEN FOR EXTEND MODE
UFBF2SHARED       EQU   X'04'            TO OPEN FOR SHARED MODE
UFBF2DALT         EQU   X'02'            DELETIONS IN PROGRESS
*                                         ON ALT-INDEXED FILE
UFBF2PLOG         EQU   X'01'            FILE PROLOGUE PRESENT
*
         EJECT
UFBDEVCLASS       DS    BL1              DEVICE CLASS (REQUIRED
*                                        BY 'OPEN')
UFBDEVCLASSWS     EQU   X'01'            WORKSTATION
UFBDEVCLASSTAPE   EQU   X'02'            MAGNETIC TAPE
UFBDEVCLASSDISK   EQU   X'03'            DISK
UFBDEVCLASSPRT    EQU   X'04'            PRINTER
UFBDEVCLASSTC     EQU   X'05'            TC DEVICE
UFBDEVCLASSDUMM   EQU   X'FF'            DUMMY FILE
UFBFLAGS          DS    BL1              FILE ATTRIBUTE FLAGS
UFBFLAGSUPDAT     EQU   X'80'            FILE HAS BEEN CLOSED
UFBFLAGSCOMP      EQU   X'40'            DATA RECORDS IN COMPRESSED
*                                        FORMAT
* ******* UFBFLAGSRECOV - RECOVERY=YES FOR BIT = ZERO ***********
UFBFLAGSRECOV     EQU   X'20'            USE PREFORMAT AND RECOVERY
*                                        PROCEDURES IF ZERO (INDEXED ONLY)
UFBFLAGSALTX      EQU   X'10'            ALTERNATE INDICES IN FILE
UFBFLAGSLOG       EQU   X'08'            CONSEC LOG FILE FLAG
UFBFLAGSALTP      EQU   X'08'            ALTERNATE-TREE PROCESS FLAG
UFBFLAGSPART      EQU   X'04'            PARTIAL BACKUP FILE
*                                        PROGRAM SETS BIT BEFORE OPEN OUTPUT
*                                        (BAM OR PAM) TO SET BIT IN FILE
*                                       LABEL, OR SETS BIT BEFORE NON-OUTPUT
*                                       OPEN (BAM OR PAM) IF ABLE TO PROCESS
*                                       PARTIAL FILES. INVALID FOR RAM.
UFBFLAGSXLCLS     EQU   X'02'            SHARED FILE EXCLUSIVE
*                                        LOCK ON CLOSE FLAG
.*** note: UFBFLAGSADMS  equ x'02'  ADMS DISK FILE INDICATOR
```

```
UFBFLAGSPRIV       EQU    X'01'              PROGRAM FILE CARRIES
*                                            ADDITIONAL ACCESS PRIVILIGES
UFBDEVADDR         DS     HL1                DEVICE ADDRESS (FOR PRINTERS
*                                            AND WORKSTATIONS ONLY.
*                                            USED IF SUPPLIED
*                                            AND PLACED HERE BY 'OPEN' IF
*                                            NOT SUPPLIED. HEX FF IF
*                                            NOT SUPPLIED.)
UFBF3              DS     0BL1 (*     NAME   KEPT FOR COMPATIBILITY *)

UFBSLOTSIZE        DS     H                  RELATIVE FILE SLOT SIZE
                             ORG UFBF3
UFBPRTCLASS        DS     CL1                PRINT CLASS (A-Z)
*
UFBFORMNO          DS     HL1                PRINTER FORM NUMBER (BINARY)
UFBPRNAME          DS     CL8                PARAMETER REFERENCE NAME
*                                            (MUST ALWAYS BE SUPPLIED HERE
*                                            FOR 'OPEN')
UFBVOLSER          DS     CL6                VOLUME SERIAL NUMBER FOR
*                                            VOLUME-ORIENTED FILES (TAPE
*                                            OR DISK)
*                                            (IF 6 ASCII BLANKS, TAKEN FROM
*                                            PROCEDURE SPECIFICATION OR
*                                            'OPEN'-TIME PROMPT. IF SPECIFIED
*                                            IN NEITHER OF THESE WAYS,
*                                            TAKEN FROM DEFAULT IN
*                                            ETCB)
UFBDIRNAME         DS     CL8                DIRECTORY NAME (IF 8 ASCII
*                                            BLANKS, DIRECTORY NAME TAKEN
*                                            FROM PROCEDURE SPECIFICATION
*                                            OR 'OPEN'-TIME PROMPT.
*                                            IF SPECIFIED IN NEITHER PLACE
*                                            AND VOLUME SERIAL ALSO
*                                            OMITTED, DEFAULT IN ETCB
*                                            USED)
         EJECT
UFBFILENAME        DS     CL8                FILE NAME (UNDER DIRECTORY)
*                                            (IF 8 BLANKS, FILE NAME TAKEN
*                                            FROM PROCEDURE SPECIFICATION
*                                            OR 'OPEN'-TIME PROMPT.
*                                            WORK FILE SPECIFICATION IF
*                                            ASCII '#' OR '$' FOLLOWED BY
*                                            FOUR ALPHAMERICS - LAST
*                                            3 CHARACTERS THEN MUST BE
*                                            BLANKS - SEE WORK FILE
*                                            DOCUMENTATION)
UFBFPCLASS         DS     CL1                FILE PROTECTION CLASS
*                                            VALUE TO LABEL IF OUT-MODE;
*                                            TAKEN FROM USER 'SET' DEFAULTS IF
*                                            X'00' IS SUPPLIED;
*                                            VALUE FROM LABEL IF EXISTING FILE
UFBCREATOR         DS     CL3                FILE CREATOR FOR NEW OR
*                                            EXISTING DISK FILES
```

```
UFBALTCNT          DS    OBL1           COUNT OF ALTERNATE INDICES
*                                       IN FILE AFTER SVC OPEN
UFBALTPTR          DS    A              POINTER TO AXD1-AREA FOR DMS
*                                       PROCESSING (ALL REFERENCE TO THE
*                                       AXD1-AREA MUST USE UFBALTPTR)
*
* FOR CONSEC FILES, THE ALTPTR FIELD HOLDS LOGICAL RECORD COUNT
                               ORG UFBALTPTR
UFBLOGRECCNT       DS    F              LOGICAL RECORD COUNT FOR START END
* FOR RELATIVE FILES, THE ALTPTR FIELD HOLDS CURRENCY INFORMATION
                               ORG UFBALTPTR
UFBRELPOS          DS    F              RELATIVE FILE LOGICAL CURRENCY PTR
* FOR DEVICES OTHER THAN DISK, THE ALTCNT FIELD IS FOR MICROCODE TYPE
                               ORG UFBALTCNT
UFBMCTYPE          DS    XL1            DEVICE TYPE
UFBMCTYPE2780      EQU   X'01'          2780 BATCH TC
UFBMCTYPE3780      EQU   X'02'          3780 BATCH TC
UFBMCTYPETCD       EQU   X'03'          TC DIAGNOSTICS
*
* FOR TC2780, TC3780 FILES, THE ALTPTR FIELD IS USED FOR THE TC
* BATCH STREAM OPTIONS
UFBTCDATAOPT       DS    BL1            TC STREAM DATA OPTION
UFBTCXMITOPT       DS    BL1            TC STREAM TRANSMIT/RECEIVE
*                                       OPTION
UFBTCMAXRECSZ      DS    XL1            TC STREAM MAXIMUM RECSIZE
*                                       MINUS 1
* FOR WORD PROCESSING WORKSTATIONS, THE ALTPTR FIELD IS USED FOR
* EXTENDED WS-ATTENTION INFORMATION
                               ORG UFBALTPTR+1
UFBWPAID           DS    XL3            EXTEND WS-ATTN INFORMATION
**
UFBF4              DS    BL1            ADDITIONAL DEVICE-DEPENDENT
*                                       FLAGS
UFBF4NOVTOC        EQU   X'80'          UNSTRUCTURED DISKETTE
UFBF4RLSE          EQU   X'40'          RELEASE UNUSED SPACE
*                                       ON CLOSE
UFBF4BLKAL         EQU   X'20'          ALLOCATE SPACE FOR NEW
*                                       DISK FILE IN BLOCKS,
*                                       FROM UFBNBLKS
UFBF4VERIFY        EQU   X'10'          VERIFY OPTION ON ALL
*                                       DISK WRITES
UFBF4NOMSG         EQU   X'08'          NO RESPECIFY OR CANCEL
*                                       MESSAGE FOR SVC OPEN
*                                       ALSO NO CANCEL ON CLOSE; NO
*                                       ACK/CANCEL FOR DMS.
UFBF4NOACK         EQU   X'04'          NO EXCEPTIONAL CONDITION
*                                       ACKNOWLEDGMENT MESSAGES
*                                       FOR DMS FUNCTIONS
UFBF4PMSG          EQU   X'02'          FOR INTERNAL USE BY DMS -
*                                       CLOSE SENDS MESSAGE TO
*                                       UNSPOOLER IF SET
```

```
UFBF4ALLOWT        EQU   X'01'            USED BY SVC OPEN. PROGRAM
*                                         SUPPLIES BIT=1 TO ALLOW DEV=TAPE.
*                                         (OPEN SETS=1 IF UFBDEV=TAPE ALSO)
*                                         OTHERWISE, DEV=TAPE NOT ACCEPTED.
UFBNRECS           DS    FL3              NUMBER OF DATA RECORDS IN
*                                         FILE (EXAMINED BY 'OPEN' FOR
*                                         OUTPUT OPEN MODE ONLY.
*                                         EXCLUDES INDEX RECORDS, ETC)
UFBNRECSUPDAT      EQU   X'80'            HI BIT SET IN NRECS HI
*                                         BYTE (RETURNED BY LOCK)
*                                         IF ON IN OFB AT LOCK TIME
UFBLRECSAVE        DS    H                RECSIZE SAVED HERE
*                                         BY OPEN (BAM)
UFBRETPD           DS    H                RETENTION PERIOD IN DAYS
*                                         (MAXIMUM 999)
UFBLOCEND          EQU   *
UFBLOCLENGTH       EQU   (UFBLOCEND-UFBBEGIN)
                    EJECT
* ************************************************************
* DATA MANAGEMENT SYSTEM SECTION
* ************************************************************
UFBBCB1            DS    BL16             BUFFER CONTROL BLOCK
*                                         (CORRESPONDS TO SVC XIO PARAMETER
*                                         LIST)
                            ORG UFBBCB1
UFBXIOFLAGS        DS    0BL1             FLAG BYTE FOR SVC XIO
UFBXIOFLAGSRLS     EQU   X'80'            RELEASE BUFFER AFTER WRITE
UFBOFB             DS    A                OFB ADDRESS
UFBBUFCMD          DS    0BL1             COMMAND BYTE FOR OPERATION
UFBBUFADR          DS    A                BUFFER MEMORY ADDRESS
*                                         (BLOCK ADDRESS WITHIN
*                                         BUFFER IF BUFFER LARGER
*                                         THAN 2K)
UFBBUFDATAL        DS    H                LENGTH IN BYTES FOR
*                                         OPERATION
UFBBUFOFFSET       DS    H                OFFSET OF NEXT RECORD
*                                         IN BUFFER
UFBBUFBLOCK        DS    FL3              (STARTING) BLOCK WITHIN
*                                         FILE OF BUFFERED DATA
UFBBCBFLAGS        DS    BL1              FLAGS
UFBBCBFLAGSLOD     EQU   X'01'            BUFFER CONTENTS VALID
UFBBCBFLAGSTOR     EQU   X'02'            BUFFER TO BE REWRITTEN
UFBBCBFLAGSIO      EQU   X'04'            BUFFER I/O IN PROGRESS
UFBBCBFLAGSPROT    EQU   X'10'            BUFFER IN PROTECTED MEMORY
UFBBCBFLAGSEOB     EQU   X'20'            END OF BLOCK REACHED
UFBBCBFLAGSEOF     EQU   X'40'            EOF BLOCK IN BUFFER
**
* THE FOLLOWING FIELDS ARE USED FOR THE TIME-OUT OPTION IN SHARED
* MODE ONLY.
                            ORG UFBBUFDATAL
UFBTIMEEXIT        DS    A                EXIT ADDRESS FOR TIME-OUT
*                                         RETURN (0 = NO TIME-OUT)
```

```
UFBHOLDID          DS    CL3           INITIALS OF HOLDER OF
*                                      RESOURCE
UFBTIME            DS    XL1           WAIT TIME IN SECONDS
*                                      (0 = NO WAIT)
**
* THE FOLLOWING FIELDS ARE USED TO RETURN STATUS INFORMATION FROM THE
* SHARER WHEN USER'S OPEN OF A SHARED FILE FAILS WITH FILE STATUS '60'
* AND AN OPEN ERROR CODE OF 'E029'.
                           ORG UFBBUFDATAL
UFBSHROPNCODE      DS    CL4           SHARER'S OPEN ERROR MSG #
UFBSHROPNRCSZ      DS    XL2           TRUE FILE RECORD SIZE
UFBSHROPNFORG      DS    X             TRUE FILE ORGANIZATION BYTE
*                                       (AS PER UFBFORG)
UFBSHROPNSPARE     DS    X             UNUSED
**
UFBBUFSIZE         DS    H             BUFFER SIZE
UFBCHKSIZE         DS    H             RESIDUAL COUNT FROM XIO
*                                      (DMS USE ONLY)
* UFBXDATE OR UFBOUTRECS IS AVAILBLE AFTER SVC OPEN AND BEFORE THE
* FIRST DMS REQUEST; UFBRES3 IS AN INTERNAL DMS FIELD AFTERWARDS.
UFBRES3            DS    BL3           RESERVED FOR INTERNAL DMS
         ORG UFBRES3
UFBXDATE           DS    BL3           EXPIRATION DATE (EXIST FILE)
         ORG   UFBRES3
UFBOUTRECS         DS    FL3           NUMBER OF RECORDS REQUESTED
*                                      FOR OUTPUT MODE
         EJECT
UFBNBLKS           DS    FL3           NUMBER OF 2048-BYTE BLOCKS
*                                      IN THE FILE
         ORG   UFBNBLKS
UFBDMSGID          DS    BL3           STORED MSG-ID(DMS NOMSG EXIT)
UFBMAXTFR          DS    H             MAXIMUM DATA TRANSFER IN
*                                      BYTES FOR DISK (SET BY OPEN)
         ORG   UFBMAXTFR
UFBRES1            DS    BL1           FUTURE SPARE BYTE
UFBOPFLAGS         DS    BL1           INTERNAL OPEN FLAGS
UFBOPFLAGSPFA      EQU   X'80'         PRINT-FILE ASSIGNMENT TO DISK
UFBOPFLAGSPFS      EQU   X'40'         PF - USER SUPPLIED FILE NAME
UFBOPFLAGSWKA      EQU   X'20'         WORK-FILE ASSIGNMENT BY OPEN
UFBOPFLAGSPVS      EQU   X'10'         PF - USER SUPPLIED VOLUME
UFBOPFLAGSSCAN     EQU   X'08'         IN SCAN BIT (WORK/SPOOL)
**
UFBLF              DS    BL1           LAST FUNCTION PERFORMED
UFBLFOPEN          EQU   X'00'         OPEN
UFBLFREAD          EQU   X'04'         READ
UFBLFWRITE         EQU   X'08'         WRITE
UFBLFREWRITE       EQU   X'0C'         REWRITE
UFBLFDELETE        EQU   X'10'         DELETE
UFBLFSTART         EQU   X'14'         START
UFBLFCLOSE         EQU   X'18'         CLOSE
UFBLFMOD           DS    BL1           LAST FUNCTION MODIFIER
*                                      (DOESN'T CHANGE ON 'REWRITE')
*                                      (SEE UFBV ABOVE)
```

```
         ORG   UFBLFMOD
UFBXCODE          DS    BL1           EXTENDED OPEN EXIT CODE
* UFBXCODE VALUES 1-8 SET FOR POSSESSION CONFLICT
UFBXCODENOINFO    EQU   X'00'         NO FURTHER INFORMATION
UFBXCODEUSE       EQU   X'01'         DEVICE IN USE
UFBXCODEDET       EQU   X'02'         DEVICE DETACHED
UFBXCODEVOLX      EQU   X'03'         VOLUME EXCLUSIVE
UFBXCODEPOSS      EQU   X'04'         FILE POSSESSION CONFLICT
UFBXCODEPAGE      EQU   X'05'         PAGING FILE - SYSTEM ONLY
UFBXCODEIMAG      EQU   X'06'         IMAGE FILE (INPUT MODE ONLY)
UFBXCODEAOPEN     EQU   X'07'         ALREADY OPEN - THIS USER
UFBXCODEAUSE      EQU   X'08'         ALREADY IN USE - THIS USER
*
* UFBXCODE VALUES X'10' - X'1F' SET FOR OPEN FORMAT ERROR
UFBXCODETRACK     EQU   X'10'         PROGRAM REQUIRES 7 TRACK
*                                     TAPE WHILE DRIVE IS 9 TRACK
*                                     OR VICE VERSA
UFBXCODEDNPRT     EQU X'11'           UFB FORG=PRINT, WHILE
*                                     FDR FORG NOT= PRINT
UFBXCODEDNPRG     EQU X'12'           UFB FORG=PROG, WHILE
*                                     FDR FORG NOT= PROG
UFBXCODEDNCSC     EQU X'13'           UFB FORG=CONSEC, WHILE
*                                     FDR FORG NOT= CONSEC
UFBXCODEDNWP      EQU X'14'           UFB FORG=WP, WHILE
*                                     FDR FORG NOT= WP
UFBXCODEDNINX     EQU X'15'           UFB FORG=INDEXED, WHILE
*                                     FDR FORG NOT= INDEXED
UFBXCODEDFGR      EQU X'16'           UFB FORG NEITHER CONSEC
*                                     NOR INDEXED---ERROR
UFBXCODENREL      EQU X'17'           UFB FORG=REL, WHILE
*                                     FDR FORG NOT= REL
UFBEREC           DS    H             LAST RECORD NUMBER WITHIN
*                                       LAST BLOCK
UFBVERSION        DS    HL1           UFB VERSION NUMBER *******
UFBEBLK           DS    FL3           LAST BLOCK NO. WITHIN FILE
*                                     FROM 0
UFBBUFSTART       DS    A             BUFFER MEMORY ADDRESS;
*                                     BUFFER CONTROL TABLE
*                                     ADDRESS BEFORE 'OPEN'
*                                     IF BUFFER POOLING
*                                     SPECIFIED (UFBF1POOL SET)
UFBRDLTH          DS    H             LENGTH IN BYTES OF
*                                     DATA IN BUFFER
UFBPRTCOPIES      DS    H             NUMBER OF PRINT COPIES
*                                     (FOR PRINTER FILES ONLY)
                          ORG UFBPRTCOPIES
UFBWPBLKSIZE      DS    X             WORD PROCESSING FILE CONTROL
UFBWPBLS          DS    X             FIELDS, WP FILES BLKSIZE
*                                     AND BYTES IN LAST SECTOR
                          ORG UFBBUFSTART
UFBPTRB           DS    FL4           FIRST BLOCK IN INDEX
*                                     AREA OF PRIMARY EXTENT
*                                     (INDEXED FILES)
```

```
UFBPTRC           DS    FL4            LAST BLOCK IN INDEX AREA
*                                      OF PRIMARY EXTENT
*                                      (INDEXED FILES)
UFBDMSEND         EQU   *
UFBDMSLENGTH      EQU   (UFBDMSEND-UFBBEGIN)
* ***********************************************************
* END OF UFB FOR ALL FILES/DEVICES EXCEPT TAPE FILES, INDEXED DISK
* FILES, and DMS/TX disk files.
                   EJECT
* ***********************************************************
* INDEXED DISK FILE EXTENSION SECTION:
* UFBKEYPOS AND UFBKEYSIZE SHOULD BE FILLED IN BY THE PROGRAM BEFORE
* 'OPEN' FOR A NEW INDEXED FILE (UFBF2OUT AND UFBFORGINDEXED SET).
* THEY ARE SET BY 'OPEN' FOR AN EXISTING INDEXED FILE. 'OPEN'
* WILL SET UFBGKSIZE TO ZERO. THE USER'S PROGRAM MAY SET IT NON-ZERO
* BEFORE A 'START' FUNCTION. 'START' WILL ZERO IT AGAIN. THE
* USER'S PROGRAM MUST NOT MODIFY ANY OTHER FIELDS THAN
* UFBGKSIZE IN THIS SECTION WHILE THE FILE IS OPEN.
* ***********************************************************
UFBKEYPOS         DS    H              KEY POSITION IN LOGICAL RECORD
UFBKEYSIZE        DS    HL1            KEY SIZE IN BYTES
UFBGKSIZE         DS    HL1            GENERIC KEY LENGTH OVERRIDE
*                                      MAY BE SET BEFORE 'START';
*                                      USED ONLY BY 'START' FUNCTION;
*                                      RESET TO BINARY 0 BY 'OPEN' AND
*                                      EVERY 'START' FUNCTION
UFBHXBLK          DS    FL3            HIGHEST-LEVEL INDEX BLOCK
*                                      ADDRESS FOR KEYED ACCESS
UFBDABLK          DS    FL3            FIRST DATA BLOCK ADDRESS
UFBPKI            DS    H              INDEX ITEMS PER BLOCK
*                                      FOR OUTPUT MODE
UFBPTRD           DS    FL4            FIRST BLOCK BEYOND
*                                      PRIMARY EXTENT
*                                      (INDEXED FILES)
UFBPTRI           DS    F              NEXT AVAILABLE INDEX
*                                      BLOCK WITHIN PRIMARY EXTENT
*                                      INDEX AREA
UFBPTRN           DS    F              NEXT AVAILABLE INDEX
*                                      OR DATA BLOCK IN A SECONDARY
*                                      EXTENT (INITIALLY ZERO)
*
               ORG UFBHXBLK
UFBSHRAXD1        DS    XL20           partial AXD1 area for shared
*                                      alternate indexed files
*
UFBBCBIOUT        DS    BL16           BCB FOR INDEX CREATION,
*                                      OUTPUT MODE
*
*        DMS/TX Before Image control area for shared indexed files
*        (internal system use only)
               ORG UFBBCBIOUT
UFBBIRECAREA      DS    A              Before Image Recarea Address
UFBBIRECSIZE      DS    H              Before Image Record Size
```

```
UFBBIAXD1MASK      DS    BL2               Before Image Record AXD1 Mask
*
                   DS    8X                RESET ASSEMBLY COUNTER
UFBPKD             DS    H                 RECORDS PER BLOCK FOR
*                                          OUTPUT MODE
UFBSPAREINX        DS    XL2               (RESERVED)
UFBINXDISKEND      EQU   *
UFBINXDISKLGTH     EQU (UFBINXDISKEND-UFBBEGIN)
                    EJECT
* ***********************************************************
* DMS/TX DISK FILE EXTENSION SECTION:
*
*  Existence of this extension section is determined by
*  UFBVERSION = 2 or greater and UFBDEVCLASSDISK set.
*
*  Input fields to the Open SVC are:
*      UFBDXOM       - Open Modifiers
*      UFBDXRECBLCK - controls Recovery Block allocation in Output
*                     mode only
*      UFBDXSPARE   - must be zero
*
*  All other fields are returned by a successful Open; input values
*  are ignored.
*
* ***********************************************************
UFBDXOM            DS    X                 DMS/TX Open Modifier Flags
*
*    Modifiers for general use on ANY disk file. (Their use is NOT
*    restricted to files under DMS/TX).
*
UFBDXOMNOMODVOL    EQU   X'80'             No modification of Volume
*                                           in Open getparms.
UFBDXOMNOMODLIB    EQU   X'40'             No modification of Library
*                                           in Open getparms.
*                                          Open exit for xlib must be set
*                                           (except output mode).
UFBDXOMNOMODFIL    EQU   X'20'             No modification of Filename
*                                           in Open getparms.
*                                          Open exit for xfile must be set.
*
UFBDXOMCKACCESS    EQU   X'10'             Restrict user access rights
*                                          to logon privileges (ignore
*                                          special program privileges)
*
UFBDXOMNOACK       EQU   X'08'             suppress acknowledge
*                                          getparms in OPEN
*
```

```
*    Modifiers for system use only for DMS/TX files opened in non-
*    shared modes.
*    Warning: Improper use can compromise the integrity of a file.
.*   users: DMSTX utility, TXPATCH utility, @SHARER@, BUILDALT, WV82
*
UFBDXOMREORGKEY   EQU   X'04'           If file requires reorg, set
*                                        UFBDXREORGLF, UFBKEYAREA to
*                                        incomplete function values
UFBDXOMNOREC      EQU   X'02'           No Recovery
UFBDXOMNOCHK      EQU   X'01'           No Check for file softcrash
*                                        or reorganization required
*
UFBDXRECBLK       DS    C               Recovery Blocks flag
*   Output mode: set to RECBLKALLO to allocate Recovery Blocks
*                set to RECBLKNO to not allocate Recovery Blocks
*   Value is returned for all other modes.
UFBDXRECBLKNO     EQU   C'N'            No Recovery Blocks
UFBDXRECBLKALLO   EQU   C'A'            Recovery Blocks Allocated
*                                       but not used
UFBDXRECBLKUSED   EQU   C'U'            Recovery Blocks allocated &
*                                       used (file is under DMS/TX)
*
UFBDXDBNAME       DS    CL6             Database Name
*
UFBDXFV#          DS    0XL12           File version #
UFBDXFV#SEQ#      DS    F               sequence #
UFBDXFV#DT        DS    PL8             date/time stamp
*
UFBDXRECO         DS    C               Recovery option after Open
                                        (usually the Database option)
                                        Not an input parameter.
UFBDXRECONO       EQU   C'N'            No Recovery
UFBDXRECOSOFT     EQU   C'S'            Softcrash Recovery
*
* ***********************************************************
* The following fields are for internal system use only:
*
UFBDXLSBREC       DS    X               LSB File Recovery Option
*
UFBDXREORGLF      DS    X               UFBLF value from incomplete
*                                       function (if UFBDXOMREORGKEY
*                                       set and file requires reorg)
*
UFBDXCS           DS    X               Crash Status of DMS/TX files
*                                       (input mode or DXOMNOCHK set)
UFBDXCSSOFT       EQU   X'01'           Softcrash Recovery required
UFBDXCSREORG      EQU   X'02'           Reorganization required
*
UFBDX#BIJS        DS    H               # BIJS accessing crashed file
*                                       (if DXOMNOCHK set)
*
UFBDXFLAGS        DS    X               Extra flag bits
UFBDXFLAGSTXON    EQU   X'80'           Turn on dmstx locking
*                                       protocol on file OPEN
```

```
UFBDXFLAGSRDNLY   EQU   X'20'         Open for shared read-only
*
UFBDXSPARE        DS    XL9            *(reserved - must be zero)
*
UFBDXEND          EQU   *
UFBDXLGTH         EQU   (UFBDXEND-UFBBEGIN)
                   EJECT
* ************************************************************
* MAGNETIC TAPE FILE EXTENSION SECTION:
* FIELDS UFBTLABELS, UFBTDEN, UFBTSEQ AND UFBTFLAGS MAY BE SET
* BEFORE 'OPEN' TO REQUEST OUTPUT LABELING OPTIONS, DENSITY
* AND FILE POSITIONING.
* ALL RELEVANT FIELDS AND FLAGS NOT SET BEFORE 'OPEN' ARE SUPPLIED
* HERE BY 'OPEN' PROCESSING AND MAY BE EXAMINED BY THE USER'S
* PROGRAM.
* ************************************************************
                           ORG UFBDMSEND
UFBTSPARE1        DS    BL4           (RESERVED)
UFBTBCB           DS    BL16          ADDITIONAL BUFFER CONTROL
*                                      BLOCK FOR TAPE DOUBLE
*                                      BUFFERING
UFBTLABELS        DS    BL1           REQUESTED LABELING (OUTPUT)
*                                      OR LABEL TYPE ON TAPE
*                                      (INPUT)
UFBTLABELSNL      EQU   X'01'          UNLABELLED
UFBTLABELSANY     EQU   X'02'         ANY TYPE OF LABEL
UFBTLABELSAL      EQU   X'04'         ASCII LABELS
UFBTLABELSIL      EQU   X'08'         IBM LABELS
UFBTDEN           DS    BL1           TAPE DENSITY
UFBTDEN800        EQU   X'01'         800 BPI
UFBTDEN1600       EQU   X'02'         1600 BPI
UFBTDEN556        EQU   X'03'         556 BPI
UFBTDEN6250       EQU   X'08'         6250 BPI
UFBTDEN6400       EQU   X'10'         6400 BPI
*
UFBTSEQ           DS    H             TAPE FILE SEQUENCE NUMBER
*                                     (SET BEFORE OR DURING
*                                     OPEN TO REQUEST POSITIONING
*                                     AND AVAILABLE AFTER OPEN)
UFBTFLG           DS    BL1           TAPE-RELATED FLAGS
UFBTFLGALLOWNL    EQU   X'80'         *** OBSOLETE ***
UFBTFLGSWITCH     EQU   X'40'         TAPE VOLUME SWITCH REOPEN
*                                     IN PROGRESS
UFBTFLGEODEOV     EQU   X'20'         TAKE EOV1 TRAILER LABEL AS
*                                      EOF1 LABEL
UFBTFLG7TRACK     EQU   X'10'         USE 7 TRACK TAPE DRIVE FOR
*                                      THIS FILE
UFBTFLGNOHDR2     EQU   X'08'         NO HDR2 FILE LABEL
UFBTVOLSEQ        DS    BL1           TAPE VOLUME SEQUENCE NUMBER
*                                      (ORDER OF VOLUME IN A
*                                      MULTIPLE VOLUME FILE)
UFBTSAVEVOL       DS    CL6           VOLUME NAME OF FIRST
*                                      VOLUME OF A MULTI-VOLUME
*                                      FILE SAVED HERE
```

```
UFBTPARITY         DS    BL1            TAPE PARITY (7 TRACK TAPE
*                                        ONLY)
UFBTPARITYODD      EQU   X'01'          ODD PARITY
UFBTPARITYEVEN     EQU   X'02'          EVEN PARITY
UFBTSPARE2         DS    BL11           (RESERVED - MUST BE 0)
UFBTAPEEND         EQU   *
UFBTAPELGTH        EQU   (UFBTAPEEND-UFBBEGIN)
         SPACE  2
.* *****************************************************************
.* ADMS DISK FILE EXTENSION SECTION was here
.* *****************************************************************
.* (ADMS) RESTART DISK FILE EXTENSION SECTION was here
.* *****************************************************************
.*
                            ORG UFBDXEND
UFBEND             EQU   *
UFBLGTH            EQU   (UFBEND-UFBBEGIN)
```

A.2 THE AXD1 DSECT

```
*********************************************************************
*
*   THE ALTERNATE INDEX DESCRIPTOR BLOCK (AXD1) DESCRIBES THE
*   ALTERNATE INDEX STRUCTURES OF AN INDEXED FILE. AN INDEXED
*   FILE HAS AN AXD1 BLOCK IF AND ONLY IF FLAG FDR1FLAGSALTX
*   IS SET IN ITS LABEL (FDR1). THE AXD1 BLOCK CONTAINS
*   UP TO 16 (64) ALTERNATE INDEX DESCRIPTIONS (AXD1ENTRY). THE
*   NUMBER OF DESCRIPTIONS IS CONTAINED IN FDR1ALTXCNT OF THE
*   FDR1 RECORD.
*
*   THE AXD1 IS LOCATED IN BLOCK NUMBER ZERO OF THE FILE.
*   THE AXD1 IS DIVIDED INTO 4 AREAS:
*            1. BLOCK DESIGNATOR AREA (AXD1BL)
*            2. DMS PROCESSING AREA (AXD1MASK TO AXD1ENTRY)
*            3. AXD ENTRIES (ONE AXD ENTRY PER ALT-INDEX)
*            4. SPARE AREA (UP TO END OF 2K BLOCK)
*   AREAS 1-3 ARE HELD IN THE AXD1-AREA  (POINTED TO BY UFBALTPTR)
*   DURING FILE PROCESSING.
*
*   DATE 07/16/82
*   VERSION 5.04.02
*
*********************************************************************
* BLOCK DESIGNATOR AREA:

AXD1BEGIN         DS    0F
AXD1BL            DS    BL4          BLOCK TYPE DESIGNATION
*                                    AXD1BL MUST EQUAL XL4'2'
*                                    OR XL4'4'
* DMS PROCESSING AREA:
AXD1MASK          DS    BL8          BITS ON INDICATE ALTERNATE
*                                    INDEX STRUCTURES (NUMBERED
*                                    1 TO 16) PRESENT
*                                    (INITIAL IMPLEMENTATION OF
*                                    2-BYTE MASK ONLY)
AXD1UFB           DS    A            POINTER TO UFB FOR THIS FILE
*                                    AFTER THE FILE HAS BEEN OPENED
AXD1ALTINX        DS    BL1          ORDINAL INDEX NUMBER FOR READ
AXD1FLAGS         DS    BL1          DMS FLAG BYTE
AXD1FLAGSOK       EQU   X'80'        ALTERNATE INDEX STRUCTURES HAVE
*                                     BEEN CREATED WHEN FLAG SET
* THE FOLLOWING FLAGS ARE USED FOR DMS PROCESSING (0 IN LABEL)
AXD1FLAGSOPENA    EQU    X'08'       OPEN ALLOCATED THIS AXD1 BLOCK
*                                     (ONLY IF NOT OUTPUT MODE)
AXD1FLAGSQ        EQU   X'04'        START QUALIFIED OPTION
AXD1FLAGSTYPER    EQU   X'02'        TYPE R SAVEAREA IN USE
AXD1FLAGSTYPEV    EQU   X'01'        TYPE V SAVEAREA IN USE
**
```

```
AXD1MSIZE          DS    BL1          SIZE OF MASK PER FILE
*                                     VALUE FROM 2-8 BYTES (MUST BE 2
*                                     FOR FIRST IMPLEMENTATION)
AXD1DUPINX         DS    BL1          ORDINAL INDEX NUMBER OF THE
*                                     ALT-TREE HAVING DUPLICATED KEY
* MINIMUM AXD1-AREA FOR SHARED MODE ENDS HERE.
* AXD1MASK, AXD1MSIZE, AND AXD1ALTINX ARE REQUIRED.
*
AXD1BCB            DS    BL16         BCB FOR DMS PROCESSING (SEE UFB)
AXD1PMASK          DS    BL8          MASK OF VALID ALTERNATE ACCESS
*                                     PATHS (SET AT FILE CREATION ONLY)
*
* THE FOLLOWING FIELDS ARE INTERMEDIATE OUTPUT MODE FIELDS
*
AXD1ORECSIZE       DS    H            WORK RECORD - MAX LENGTH
AXD1OFLAGS         DS    BL1          OUTPUT FLAGS (RESERVED)
AXD1OSTART         DS    BL3          FIRST BLOCK CONTAINING WORK RECORDS
AXD1ONRECS         DS    BL3          TOTAL COUNT OF WORK RECORDS
AXD1OEBLK          DS    BL3          LAST USED BLOCK NUMBER IN PRIMARY
*                                     TREE (ALT-TREE TO AXD1EBLK+1)
AXD1OSPAREX        DS    H            **** (unused) ****
AXD1OSPARE         DS    BL2          RESERVED IN OUTPUT MODE
**
                      ORG AXD1ORECSIZE
* THE FOLLOWING FIELDS ARE USED FOR DMS PROCESSING (EXISTING FILES)
**
AXD1SAVEADR        DS    A            SAVE AREA ADDRESS (TYPE V)
AXD1SAVELTH        DS    H            SAVE AREA LENGTH (TYPE V)
                      ORG AXD1ORECSIZE
* THE FOLLOWING 3 FIELDS ARE USED FOR SAVE AREA TYPE S
AXD1SKEYSIZE       DS    BL1          SAVED PRIMARY KEYSIZE
AXD1SHXBLK         DS    BL3          SAVED PRIMARY ROOT BLOCK NUMBER
AXD1SEREC          DS    H            SAVED PRIMARY LEVEL COUNT
*
AXD1ENTOFF         DS    H            OFFSET OF ACTIVE AXD1ENTRY(IN AXD1)
AXD1PTRN  DS   BL3      NEXT SEQUENTIAL BLOCK (ALT-TREE)
AXD1CURINX         DS    BL1          ORDINAL NUMBER ASSOCIATED WITH
*                                     BLOCK IN AXD1BCB
AXD1SPAREX         DS    H            **** (unused) ****
AXD1EXSPARE        DS    BL2          SPARE - ALL FILES
**
*
*********************************************************************
* AXD1MASK AND AXD1ALTINX ARE THE ONLY FIELDS IN THE AXD1-AREA WHICH
* MAY BE MODIFIED BY THE USER-PROGRAM WHILE THE FILE IS OPEN.
*
* FOR EXISTING FILES, NO FIELDS IN THE AXD1-AREA ARE USER-SUPPLIED
* PRIOR TO ISSUING SVC OPEN.
*
* FOR OUTPUT MODE, USER-PROGRAM FILLS IN THE REQUIRED AXD1-AREA WITH:
*       AXD1MSIZE (THE ACCESS MASK PREFIX SIZE);
*       AXD1KEYPOS, AXD1KEYSIZE, AXD1EFLAGS, AND AXD1XORD
*            FOR EACH AXD1ENTRY (COUNT IN UFBALTCNT).
*********************************************************************
```

```
*
* AXD ENTRIES:
AXD1ENTRY          DS    0XL28       UP TO 64 ENTRIES
*                                    (EACH A DESCRIPTION OF ONE
*                                    ALTERNATE INDEX STRUCTURE;
*                                    UNUSED ENTRIES ZERO-FILLED)
AXD1XORD           DS    HL1         ORDINAL NUMBER  (STARTING FROM 1)
*                                    IDENTIFYING THIS INDEX STRUCTURE
*                                    (CORRESPONDS TO BIT IN
*                                    AXD1MASK)
AXD1EFLAGS         DS    BL1         OPTION FLAGS
AXD1EFLAGSDUPS     EQU   X'80'       DUPLICATE KEYS ALLOWED
AXD1EFLAGSKCOM     EQU   X'40'       KEY COMPRESSION IN INDEX
*                                     (NOT IN FIRST VERSION)
* THE FOLLOWING FLAGS ARE USED FOR DMS PROCESSING (0 IN LABEL)
AXD1EFLAGSACT      EQU   X'02'       INDICATES THIS ALT-TREE IS THE
*                                    ACTIVE ALT-TREE DURING PROCESSING
AXD1EFLAGSUP       EQU   X'01'       INDICATES AXD1PTRD, AXD1XLEVELS
*                                     OR AXD1HXBLK HAS BEEN MODIFIED
*                                     DURING ALT-TREE PROCESSING
AXD1XLEVELS        DS    H           NUMBER OF LEVELS OF THIS
*                                    ALTERNATE INDEX STRUCTURE
*                                    EXCLUDING LOWEST LEVEL
AXD1KEYPOS         DS    H           KEY POSITION IN RECORD
AXD1KEYSIZE        DS    HL1         KEY LENGTH
AXD1HXBLK          DS    FL3         BLOCK-IN-FILE OF ROOT BLOCK
*                                    OF THIS ALTERNATE INDEX
AXD1NRECS          DS    BL3         ITEM COUNT - LOW LEVEL OF TREE
AXD1PTRD           DS    FL3         FIRST BLOCK OF LOW LEVEL
*                                    OF THIS ALTERNATE INDEX
*                                    (ALTERNATE KEY SEQUENCE)
AXD1PRLEN          DS    BL1         LENGTH OF ALT TREE PSEUDO-REC
AXD1PRAKPOS        DS    BL1         POS OF ALT KEY IN PSEUDO-REC
AXD1PRPKPOS        DS    BL1         POS OF PRI KEY IN PSEUDO-REC
AXD1ESPARE         DS    BL9         (RESERVED IN EACH ENTRY)
AXD1ENTRYEND       EQU   *
AXD1ENTRYLENGTH    EQU   AXD1ENTRYEND-AXD1ENTRY
*
                           ORG AXD1ENTRY+64*L'AXD1ENTRY
AXD1SPARE3         DS    XL196       (RESERVED)
*
AXD1END            EQU   *
AXD1LENGTH         EQU   AXD1END-AXD1BEGIN
```

APPENDIX B
DMS FUNCTION REQUESTS AND MODIFIERS

B.1 RAM Function Requests and Their Modifiers

Fixed Length Records in Consecutive Files on Disk:

	Input Mode	Output Mode	IO Mode	Extend Mode	Shared Mode
READ	no mod REL NODATA		no mod REL HOLD NODATA		no mod REL HOLD
WRITE		no mod	no mod	no mod	no mod
REWRITE			no mod		no mod
START	BEGIN SKIP	EXTEND OUTPUT IO	EXTEND OUTPUT IO END BEGIN SKIP	EXTEND OUTPUT IO	HOLD HOLD,EQUAL HOLD,RANGE HOLD,LIST HOLD,RETRIEVAL RELEASE END BEGIN SKIP
DELETE					

Variable Length Records in Consecutive Files on Disk:

	Input Mode	Output Mode	IO Mode	Extend Mode	Shared Mode
READ	no mod NODATA		no mod HOLD NODATA		no mod HOLD
WRITE		no mod	no mod	no mod	no mod
REWRITE			no mod		no mod
START	BEGIN SKIP	EXTEND OUTPUT IO	EXTEND OUTPUT IO END BEGIN SKIP	EXTEND OUTPUT IO	HOLD HOLD,EQUAL HOLD,RANGE HOLD,LIST HOLD,RETRIEVAL RELEASE END BEGIN SKIP
DELETE					

Records in Relative Files on Disk:

	Input Mode	Output Mode	IO Mode	Extend Mode	Shared Mode
READ	no mod REL NODATA		no mod REL HOLD NODATA		
WRITE		no mod	no mod EOF	no mod	
REWRITE			no mod REL		
START	EQ GT GE LE LT	OUTPUT EXTEND IO	OUTPUT EXTEND IO EQ GT GE LT LE	OUTPUT EXTEND IO	
DELETE			no mod REL EOF		

Records in Indexed Files on Disk:

	Input Mode	Output Mode	IO Mode	Extend Mode	Shared Mode
READ	no mod KEYED NODATA		no mod HOLD (KEYED,HOLD) NODATA		no mod HOLD (KEYED,HOLD)
WRITE		no mod	no mod	no mod	no mod
REWRITE			no mod		no mod
START	EQ GT GE		EQ GT GE		EQ GT GE HOLD HOLD,RANGE HOLD,LIST HOLD,RETRIEVAL RELEASE
DELETE			no mod		no mod

Records in Consecutive Workstation Files:

	I/O Mode
READ	no mod MOD ALTERED TABS
WRITE	
REWRITE	no mod SELECTED TABS
START	ATTNT
DELETE	

Records in Consecutive Tape Files:

	Input Mode	Output Mode	Extend Mode
READ	no mod NEXT NODATA		
WRITE		no mod	no mod
REWRITE			
START	WAIT	WAIT	WAIT
DELETE			

OPEN	MODE
CLOSE	no modifier UNLOAD NOREWIND REEL

B.2 BAM Function Requests and Their Modifiers

	Input Mode	Output Mode	I/O Mode	Extend Mode	Shared Mode
READ	no mod REL NODATA		no mod HOLD REL NODATA		
WRITE		no mod	no mod	no mod	
REWRITE			no mod		
START		EXTEND OUTPUT IO		EXTEND OUTPUT IO	
DELETE					

B.3 PAM Function Requests and Their Modifiers

	Input Mode	Output Mode	IO Mode	Extend Mode	Shared Mode
READ	no mod		no mod		
WRITE		no mod	no mod		
REWRITE			no mod		
START	WAIT	WAIT EXTEND OUTPUT IO	WAIT		
DELETE					

APPENDIX C
DMS ERROR MESSAGES

C.1 INTRODUCTION

Appendix C contains the following types of messages, listed in the order they appear in the appendix:

- SVC OPEN Cancel Messages
- SVC OPEN Respecify Messages
- DMS Function Request Cancel Messages
- SVC CLOSE Cancel Messages
- File Status (FS) Codes for DMS

The following types of messages are not included in this appendix:

1. Messages issued by program 'BUILDALT' for OUTPUT mode creation of alternate indexed files (acknowledge and cancel messages).
2. Miscellaneous acknowledge messages from SVC OPEN and DMS function requests.

The DMS No-Message Option

The No-Message Option is available in SVC OPEN, SVC CLOSE, and DMS. This option causes the suppression of messages normally appearing on the workstation screen.

If you specify the No Message option (UFBF4NOMSG = 1), DMS sets the file status for the operation equal to C'60'. For SVC OPEN and SVC CLOSE, the message ID is stored in the first four bytes of the UFB. Return is made using the address in UFBERRAD; if this address is zero, DMS ignores the value of UFBF4NOMSG, and always displays a message.

C.2 SVC OPEN CANCEL MESSAGES

These messages deal primarily with invalid information supplied in the UFB. Some also refer to unusual conditions that rarely arise during normal SVC OPEN usage; for example, UPDATFDR SVC errors, I/O errors when reading AXD1 blocks, etc.

NOTE

There is no continuation possible when these messages are issued.

ERROR NUMBER	MESSAGE
E000	INVALID UFB ADDRESS PRESENTED TO SVCOPEN.
E001	DEVICE CLASS (XX) = INVALID OPEN MODE (XX) = INVALID FILE ORGANIZATION (XX)= INVALID RECORD SIZE = INVALID RECORDS ARE FIXED LENGTH. KEY SIZE = XXX KEY POSITION = INVALID
E002	FILE ALREADY OPEN (UFBF1OPEN SET)
E006	TASK WORKSTATION NOT AVAILABLE.
E007	MAXIMUM NUMBER OF FILES ALREADY OPEN
E011	REQUIRED BUFFER(S) NOT AVAILABLE FOR FILE PROCESSING
E014	UNEXPECTED DEALLOCATION ERROR FOR MAGTAPE DEVICE.
E016	BACKGROUND TASK ATTEMPTED TO OPEN WORKSTATION. THE WORKSTATION MAY BE OPENED IN FOREGROUND ONLY.
E017	INVALID OPEN MODE FOR PHYSICAL ACCESS METHOD. (EXTEND AND SHARED MODES ARE INVALID.)
E018	THE PROGRAM IS REQUESTING AN INVALID OPEN MODE (SHARED, EXTEND) FOR A FILE RESIDING ON AN UNSTRUCTURED DISKETTE VOLUME.
E019	THE PROGRAM IS REQUESTING AN INVALID MODE (EXTEND MODE) FOR INDEXED FILE PROCESSING.
E021	THE PROGRAM IS REQUESTING AN INVALID FILE ORGANIZATION (INDEXED) FOR A FILE RESIDING ON AN UNSTRUCTURED DISK VOLUME.
E023	INVALID ACCESS METHOD SPECIFICATION IN UFB (UFBF1).
E024	BLOCK ALLOCATION ERROR. SPACE NOT AVAILABLE ON VOLUME AND EXIT-OPTION NOT IN USE.
E025	THE PROGRAM IS REQUESTING AN INVALID MODE (SHARED MODE) FOR FILE PROCESSING UNDER THE BLOCK ACCESS METHOD (BAM).
E027	SHARING TASK NOT ACTIVE.
E028	UNABLE TO GET UNIQUE PORT NAME.
E029	SHARER RESPONSE CODE = XX-YYYY. UNEXPECTED ERROR HAS OCCURRED WHILE OPENING A FILE FOR SHARED ACCESS.
E030	INVALID BUFFER POOL SPECIFICATION. (ACCESS METHOD SUPPLIED IS INVALID.) BUFFER POOLING CAN ONLY BE USED WITH INDEXED FILES IN INPUT OR IO MODE.
E031	THE BUFFER POOL TABLE ADDRESS SUPPLIED (IN UFBBUFSTART) IS INVALID.
E032	THE BUFFER POOL TABLE HAS NOT BEEN CORRECTLY INITIALIZED.
E033	THE BUFFER COUNT SUPPLIED BY THE PROGRAM IS TOO SMALL. THE MINIMUM BUFFER COUNT IS 3.
E034	AN UNEXPECTED ERROR HAS OCCURRED WHILE UPDATING THE FILE LABEL. THE FILE CANNOT BE SUCCESSFULLY OPENED FOR UPDATE.

ERROR NUMBER	MESSAGE
E035	THE ALTERNATE INDEX BLOCK (AXD1) ADDRESS SUPPLIED IS INVALID.
E036	THE ALTERNATE INDEX COUNT IN UFBALTCNT IS INCORRECT.
E037	ALTERNATE INDEX INFORMATION FOR FILE CREATION IS INCORRECT.
E038	UNABLE TO READ AXD1 BLOCK FROM FILE BLOCK 0.
E039	FAIL TO FREE THE BUFFER AFTER READING AXD1 FROM FILE BLOCK 0.
E040	ALTERNATE INDEX KEYSIZE PLUS PRIMARY KEYSIZE TOO LARGE.
E050	UNABLE TO ALLOCATE SYSTEM MEMORY--GETMEM FAILURE.

E084	BEFORE IMAGE JOURNAL READ OPERATION HAS FAILED
E085	SYSTEM LIMIT ON THE NUMBER OF DATABASE FILES OPEN BY ONE TASK EXCEEDED.
E086	BEFORE IMAGE JOURNAL WRITE OPERATION HAS FAILED.
E087	SYSTEM ERROR - HEAP ALLOCATION HAS FAILED.
E088	TOO MANY BEFORE IMAGE JOURNALS ALREADY EXIST FOR THIS USER/DATABASE. PLEASE RECOVER THE DATABASE.
E089	THE BEFORE IMAGE JOURNAL VOLUME IS NOT MOUNTED.
E090	THE BEFORE IMAGE JOURNAL VOLUME IS OUT OF SPACE.
E091	THE BEFORE IMAGE JOURNAL VOLUME IS BEING USED EXCLUSIVELY.
E092	UNEXPECTED BEFORE IMAGE JOURNAL OPEN ERROR.
E093	BEFORE IMAGE JOURNAL RECOVERY BLOCKS INITIALIZATION ERROR.
E094	RECOVERY OPTION FILE NOT FOUND FOR THIS DATABASE. PLEASE CONTACT A RESPONSIBLE PARTY OR RUN DMS/TX UTILITY (DATABASE CREATION).
E095	RECOVERY OPTION FILE NOT FOUND. IT IS LIKELY THAT YOUR IPL VOLUME HAS BEEN CHANGED OR THAT THE DMS/TX UTILITY (DATABASE CREATION) HAS NEVER BEEN RUN.
E096	RECOVERY OPTION FILE OPEN ERROR.
E097	RECOVERY OPTION FILE READ ERROR.
E098	RECOVERY OPTION FILE CLOSE ERROR.
E099	ROLLBACK BY THIS TASK HAS FAILED ON THIS DATABASE. NO FURTHER FILES IN THIS DATABASE MAY BE OPENED BY THE PROGRAM.
E100	IO ERROR WHILE READING RECOVERY BLOCKS.
E101	UNEXPECTED RETURN CODE FROM XIO WHILE READING RECOVERY BLOCKS.
E102	SYSTEM ERROR - UNABLE TO FIND DBTB FOR COMMUNICATION TO SHARER.
E103	SYSTEM ERROR - UNABLE TO COMMUNICATE WITH THE SHARER - XMIT FAILED.
E104	UPDATFDR ERROR WHILE ATTACHING BEFORE IMAGE JOURNAL TO DATABASE.
E105	FILE MAY CONTAIN INCOMPLETE TRANSACTIONS. SOFTCRASH RECOVERY IS REQUIRED BEFORE FURTHER UPDATE ACCESS.
E106	FILE INDEX STRUCTURE IS DAMAGED. COPY/REORG IS REQUIRED PRIOR TO FURTHER UPDATE ACCESS.
E107	HEAP AREA OVERWRITTEN.
E108	UNEXPECTED ERROR WHILE UPDATING RECOVERY BLOCK.
E110	BEFORE IMAGE JOURNAL REWRITE ERROR.
E111	SPARE BYTES IN DMS/TX UFB EXTENSION MUST BE SET TO ZEROS.
E112	ILLEGAL DMS/TX OPEN MODIFIER VALUE (UFBDXOM).
E113	UFB VERSION NUMBER TOO HIGH.

ERROR NUMBER	MESSAGE
E114	RECOVERY BLOCK ALLOCATION SPECIFIED FOR UNSUPPORTED FILE ORGANIZATION.
E115	ATTEMPT TO OPEN A DMS/TX FILE BUT DMS/TX IS NOT SUPPORTED ON THIS SYSTEM. PLEASE DETACH FILE FROM DATABASE.
E116	XLIB EXIT MUST BE SET WHEN UFBDXOMNOMODLIB IS SET.
E117	XFILE EXIT MUST BE SET WHEN UFBDXOMNOMODFILE IS SET.
E118	USE OF NO-CHECK AND NO-RECOVERY OPTIONS ILLEGAL IN SHARED MODE.
E119	ILLEGAL RECOVERY BLOCK ALLOCATION VALUE SPECIFIED FOR OUTPUT MODE.
E120	SYSTEM ERROR - SHARER BUFFER SPACE EXHAUSTED.

C.3 SVC OPEN RESPECIFY MESSAGES

These messages deal with situations where the user may successfully continue either by supplying additional information or by correcting information already supplied. Situations involving possession conflicts or volume mounting are also handled by these respecification messages. The user may always continue after a respecify message.

ERROR NUMBER	MESSAGE
R001	FILE IDENTIFICATION INFORMATION IS INCOMPLETE. PLEASE SUPPLY THE MISSING INFORMATION BELOW.
R002	PLEASE SUPPLY THE APPROXIMATE NUMBER OF RECORDS IN THE DISK FILE TO BE CREATED. THIS VALUE WILL BE USED FOR INITIAL DISK-SPACE ALLOCATION.
R003	DEVICE SPECIFIED IS UNKNOWN OR NOT SUPPORTED. PLEASE RESPECIFY.
R004	DEVICE SPECIFIED IS INVALID FOR THIS PROCESSING MODE. PLEASE RESPECIFY.
R005	THE PROGRAM IS NOT REQUESTING A CONSECUTIVE-PRINT FILE. THEREFORE, THE FILE CANNOT BE ASSIGNED TO A PRINTER. PLEASE SPECIFY ANOTHER DEVICE TYPE.
R006	DEVICE NUMBER INCORRECTLY SPECIFIED. PLEASE RESPECIFY DEVICE (E.G. PRINTER 3). (NOTE--DEVICE NUMBER IS OPTIONAL AND MAY BE OMITTED).
R007	DEVICE NUMBER DOES NOT CORRESPOND TO DEVICE CLASS. PLEASE RESPECIFY DEVICE.
R008	NO PRINTER CURRENTLY AVAILABLE. ASSIGN OUTPUT TO DISK OR FREE PRINTER FOR ALLOCATION.
R013	THE FILE BELOW IS ALREADY OPENED BY THIS PROGRAM. PLEASE SPECIFY ANOTHER FILE.
R014	UNEXPECTED READFDR SVC ERROR.
R014	FILE SPECIFIED NOT FOUND IN LIBRARY. PLEASE RESPECIFY FILENAME.
R014	LIBRARY NOT FOUND IN VOLUME TABLE OF CONTENTS. PLEASE RESPECIFY LIBRARY.
R016	THE FILE SPECIFIED IS IN USE AS A SYSTEM-ONLY PAGING FILE. PLEASE RESPECIFY.
R018	THIS FILE IS CURRENTLY IN USE AS A PROGRAM FILE. THEREFORE, IT CAN ONLY BE OPENED IN INPUT MODE. PLEASE RESPECIFY THE FILE.
R020	UNEXPECTED CREATFDR SVC ERROR.
R020	FILE SPECIFIED ALREADY EXISTS. PLEASE RESPECIFY FILE.
R020	VTOC FULL, NO ROOM FOR FILE LABEL. PLEASE SPECIFY ANOTHER VOLUME.
R020	VOLUME FULL, NO ROOM FOR FILE. PLEASE SPECIFY ANOTHER VOLUME OR USE A SMALLER FILE SIZE.
R021	INVALID INFORMATION IN FILE LABEL. PLEASE RESPECIFY FILE.
R022	THE TAPE SPECIFIED BELOW IS AN NL-TAPE, BUT PROGRAM REQUIRES A TAPE WITH A DIFFERENT LABEL TYPE. PLEASE RESPECIFY.

ERROR NUMBER	MESSAGE
R024	THE FILE AT POSITION XXX WITHIN THE TAPE VOLUME IS XXXXXXXXXXXXXX. THIS DOES NOT AGREE WITH THE FILE SPECIFIED BELOW. PLEASE RESPECIFY.
R025	THE DEVICE SPECIFIED IS ALREADY IN USE BY THIS PROGRAM. PLEASE RESPECIFY.
R026	THE DEVICE SPECIFIED HAS BEEN LOGICALLY DETACHED AND IS THEREFORE NOT AVAILABLE. PLEASE RESPECIFY.
R027	THE PROGRAM REQUIRES XXXXXXXXXX. THE FILE SPECIFIED BELOW IS XXXXXXXXXX. PLEASE RESPECIFY.
R028	THE PROGRAM REQUIRES A FILE CONTAINING XXXXX-CHARACTER RECORDS. THE FILE SPECIFIED BELOW CONTAINS XXXXX-CHARACTER RECORDS. PLEASE RESPECIFY.
R029	A FILE SEQUENCE NUMBER OF ZERO IS INVALID. PLEASE RESPECIFY.
R030	TAPE IO ERROR OCCURRED DURING TAPE POSITIONING OR LABEL PROCESSING. IOSW = XXXXXXXX XXXXXXXX. PLEASE RE-MOUNT THE TAPE VOLUME IN ORDER TO TRY AGAIN.
R031	THE TAPE VOLUME IS WRITE-PROTECTED, AND THEREFORE CANNOT BE PROCESSED IN OUTPUT OR EXTEND MODE. PLEASE PUT A WRITE-ENABLE RING ON THE TAPE, AND RE-MOUNT THE VOLUME, OR USE (ENTER) TO RESPECIFY.
R032	THE UNSTRUCTURED DISKETTE VOLUME SPECIFIED FOR OUTPUT IS CURRENTLY IN USE. PLEASE RESPECIFY.
R033	THE PROGRAM IS REQUESTING A FILE THAT RESIDES ON AN UNSTRUCTURED DISK VOLUME. THE FILE SPECIFIED BELOW RESIDES ON A DISK VOLUME WITH A VTOC. PLEASE RESPECIFY.
R034	THE INDEXED FILE SPECIFIED BELOW CAN NOT BE PROCESSED IN EXTEND MODE. PLEASE RESPECIFY. (EXTEND MODE IS ONLY SUPPORTED FOR CONSECUTIVE FILES.)
R035	THE INDEXED FILE SPECIFIED BELOW WAS NOT CLOSED AT FILE CREATION. THE FILE IS CURRENTLY NOT USEABLE AND SHOULD BE RE-CREATED. PLEASE SPECIFY ANOTHER FILE.
R036	THE FILE SPECIFIED BELOW WAS NOT CLOSED AT FILE CREATION. THEREFORE, THE FILE LABEL INDICATES THAT THE FILE CONTAINS NO RECORDS. IF YOU WISH TO ACCESS THE WHOLE FILE SPACE (MAXIMUM NUMBER OF RECORDS), USE PF2 AND THE END-OF-FILE INDICATOR WILL BE SET ACCORDINGLY. OTHERWISE, PLEASE SPECIFY ANOTHER FILE.
R037	CODE = XX; UNEXPECTED OUTPUT-FILE SCRATCH ERROR. PLEASE SPECIFY ANOTHER OUTPUT FILE NAME IN ORDER TO CONTINUE.
R038	UNABLE TO FIND FILE SPACE ON ANY ELIGIBLE VOLUME. PLEASE SPECIFY A SMALLER FILE, USE A PRIVATE VOLUME, OR RELEASE (THROUGH SCRATCH) THE REQUIRED DISK SPACE.
R039	THE DISKETTE VOLUME SPECIFIED BELOW IS WRITE-PROTECTED. PLEASE RE-MOUNT THIS DISKETTE WITH WRITE-ENABLED, OR SPECIFY ANOTHER FILE.
R040	THE FILE SPECIFIED BELOW ALREADY EXISTS. USE PF3 IF YOU WISH TO SCRATCH THE EXISTING FILE AND CONTINUE. OTHERWISE, PLEASE SPECIFY ANOTHER FILE NAME.
R041	THE FILE SPECIFIED BELOW IS CURRENTLY IN USE AND CANNOT BE SCRATCHED. PLEASE SPECIFY ANOTHER OUTPUT FILE NAME.

ERROR NUMBER	MESSAGE
R045	ENTER KEY USED WITH INVALID DEVICE SPECIFICATION BELOW. USE PF4 KEY FOR MOUNT OPERATION. IF A MOUNT OPERATION IS NOT REQUIRED, PLEASE USE THE ENTER KEY WITH DEVICE = DISK.
R047	SHARER RESPONSE CODE = XX-YYYY. CONSULT SHARER ERROR LIST FOR EXPLANATION. PLEASE SPECIFY ANOTHER FILE IN ORDER TO CONTINUE.
R048	INVALID VALUE ENTERED FOR PRINTER OPTION. FORM # MUST BE LESS THAN 256. PRTCLASS MUST BE A LETTER (A-Z). COPIES MUST BE A NUMBER BETWEEN 1 AND 32,767. PLEASE RESPECIFY.
R049	THE FILE SPECIFIED BELOW IS A PROGRAM FILE WITH SPECIAL ACCESS RIGHTS. ONLY A SECURITY ADMINISTRATOR MAY MODIFY THIS FILE. PLEASE RESPECIFY.
R049	THE CURRENT USER DOES NOT HAVE THE REQUIRED ACCESS RIGHTS FOR THE FILE SPECIFIED BELOW. PLEASE RESPECIFY.
R050	THIS FILE IS A PARTIAL FILE CREATED BY BACKUP FOR USE BY RESTORE. IT MAY BE OPENED ONLY IN BAM OR PAM, WITH THE PARTIAL FILE FLAG SET. PLEASE RESPECIFY.
R051	THE SHARER HAS RUN OUT OF MEMORY FOR ITS CONTROL BLOCKS. THIS FILE MAY BE OPENED SUCCESSFULLY AFTER ENOUGH MEMORY HAS BEEN RELEASED (BY OTHER SHARED USERS).
R052	THE FILE BELOW IS ALREADY OPENED IN SHARED MODE BY THIS PROGRAM. PLEASE SPECIFY ANOTHER FILE.
R053	FILE SPECIFIED NOT FOUND IN LIBRARY. PLEASE RESPECIFY FILENAME.
R054	LIBRARY NOT FOUND IN VOLUME TABLE OF CONTENTS. PLEASE RESPECIFY LIBRARY.
R059	THE VOLUME SPECIFIED IS MOUNTED FOR EXCLUSIVE USE. A FILE ON AN EXCLUSIVE VOLUME MAY NOT BE SHARED. PLEASE SPECIFY ANOTHER FILE (OR RE-MOUNT THIS VOLUME).
R060	THE PROGRAM REQUIRES A FILE WITH A DIFFERENT FILE-ORGANIZATION FROM THE FILE SPECIFIED BELOW. PLEASE RESPECIFY.
R061	THE PROGRAM REQUIRES A FILE WITH A DIFFERENT RECORD SIZE FROM THE FILE SPECIFIED BELOW. PLEASE RESPECIFY.
R062	THE DISK VOLUME SPECIFIED IS NOT MOUNTED. PLEASE MOUNT THE DISK VOLUME OR RESPECIFY.
R063	THE FILE SPECIFIED BELOW IS CURRENTLY IN NON-SHARED USE. PLEASE RESOLVE THIS POSSESSION CONFLICT OR RESPECIFY.
R064	THE CURRENT USER DOES NOT HAVE THE REQUIRED ACCESS RIGHTS TO SCRATCH THE FILE SPECIFIED BELOW. PLEASE SPECIFY ANOTHER FILE.
R065	THE RETENTION PERIOD FOR THE FILE SPECIFIED BELOW HAS NOT EXPIRED. THE FILE CANNOT BE SCRATCHED UNLESS THE EXPIRATION DATE IS MODIFIED. PLEASE SPECIFY ANOTHER FILE OR USE THE COMMAND PROCESSOR TO MODIFY THE EXPIRATION DATE AND SCRATCH THIS FILE.
R066	THE CONSECUTIVE FILE SPECIFIED BELOW CAN NOT BE OPENED IN SHARED MODE. PLEASE RESPECIFY.
R067	THE FIRST CHARACTER OF A LOG-FILE BEING OPENED IN SHARED MODE MAY NOT BE "#". PLEASE RESPECIFY
R068	THE PROGRAM WILL NOT ACCEPT THIS FILE FROM TAPE. PLEASE RESPECIFY.

ERROR NUMBER	MESSAGE
R069	END OF TAPE REACHED WHILE POSITIONING TAPE BY FILE SEQUENCE NUMBER.
R070	VOLUME FULL, UNABLE TO ADD ANOTHER FILE ON THE TAPE. PLEASE RESPECIFY.
R071	THE TAPE FILE SPECIFIED BELOW IS NOT ON THE TAPE VOLUME. PLEASE RESPECIFY.
R072	THE DEVICE SPECIFIED IS NOT A TELECOMMUNICATION DEVICE. PLEASE RESPECIFY.
R073	CONTROL BLOCKS (PPB, LCB) FOR THIS TC DEVICE ARE NOT PROPERLY SET UP. PLEASE RESPECIFY.
R074	UNABLE TO LOAD THE MICROCODE FOR THIS TC DEVICE. PLEASE RESPECIFY.
R075	UNABLE TO CONNECT THE TC LINE, OR INCORRECT CONNECT PARAMETERS SUPPLIED. PLEASE RESPECIFY.
R076	THE PROGRAM HAS SUPPLIED AN INVALID ADDRESS FOR THE CONNECT PARAMETER. PLEASE RESPECIFY.
R077	THE TAPE VOLUME IS NOT THE CORRECT SEQUENTIAL VOLUME FOR THIS TAPE FILE. PLEASE RESPECIFY.
R078	EXTEND MODE PROCESSING FOR IBM LABELED TAPE IS NOT SUPPORTED. PLEASE RESPECIFY.
R080	THE PROGRAM HAS ATTEMPTED TO OPEN A RE-RESTART FILE, BUT THE FILE SPECIFIED IS NOT A RESTART FILE. PLEASE RESPECIFY.

C.4 DMS FUNCTION REQUEST CANCEL MESSAGES

The file status message (ID = 000) covers all file status values including cases where the significance of the FS value is determined by additional factors such as current function request and file organization. The file status message appears as a cancel message if UFBERRAD = 0. Otherwise, an acknowledge message is issued before taking the error exit. (The acknowledge message may be masked out by using UFBF4NOACK.)

Other DMS cancel messages reflect unusual conditions caused mainly by incorrect user modification of UFB fields, unexpected errors, or invalid block contents for indexed file processing.

These messages may be issued as a result of any one of the five DMS function requests or by the DMS CLOSE statement (for the last I/O operation on the file).

ERROR NUMBER	MESSAGE	POSSIBLE CAUSE
000	ERROR DETECTED AND USER ERROR EXIT NOT IN USE. FILE STATUS = XX.	Check meaning of File Status code for cause of error message. (Refer to page 354.3.)
001	INVALID FIELD FOUND WHILE PROCESSING FILE X (UFBBCBFLAGS).	Invalid buffer status flags in the UFB.
002	INVALID BLOCK NUMBER DETECTED BY SVC XIO (UFBBUFBLOCK) WHEN ATTEMPTING DISK I/O.	UFBBUFBLOCK contains invalid data. Can be caused by invalid data in Data Link Chain of the prior block.
003	INVALID FIELD FOUND WHILE PROCESSING FILE X (UFBRECSIZE=0).	Cannot have files with record lengths of zero.
004	INVALID BUFFER POOL INFORMATION DETECTED AT BEGINNING OF FUNCTION REQUEST.	There is an error in the Buffer Control Entry.
005	INVALID OFB POINTER FOUND (UFBOFB).	OFB Pointer in the UFB contains an address which is not an OFB.
006	UNEXPECTED ERROR FOUND WHILE ATTEMPTING TO ALLOCATE ADDITIONAL DISK SPACE.	SVC return code is incorrect. This is an indication of a serious DMS problem.
007	VTOC I/O ERROR OCCURED DURING SVC ALEX.	A VTOC IO error occurred while trying to obtain an additional extent. ALEX is an acronym for Allocate Extent.
008	UNABLE TO ALLOCATE DISK EXTENT SINCE ALL BUFFERS OR GETMEM POOL IN USE.	ALEX return code = 20. No work space available for UPDATFDR.
009	MAG TAPE READ OPERATION FAILED; NO DATA WAS TRANSFERRED.	Residual count greater than or equal to block size--probable IOP firmware error.
010	FUNCTION-REQUEST ISSUED ON NON-OPENED FILE.	Before performing a task within a file, the file must first be opened.
011	SECOND PHYSICAL I/O OPERATION ISSUED ON FILE WITHOUT WAITING FOR PREVIOUS I/O COMPLETION.	Occurs when two XIO's in a row were performed with no CHECK operation between them.
012	UNUSED	
013	ERROR FOUND WHILE READING FILE INDEX. THIS FILE SHOULD BE REORGANIZED IN ORDER TO GENERATE THE INDEX CORRECTLY.	Invalid condition exists in the index block currently being read.

ERROR NUMBER	MESSAGE	POSSIBLE CAUSE
014	INVALID RECORD FORMAT DETECTED. ERROR OCCURRED WHEN EXPANDING COMPRESSED DATA RECORD.	A record within a block contains garbage.
015	INVALID BLOCK NUMBER FOUND WHILE BUILDING OR UPDATING THE FILE INDEX. FILE INDEX.	Occurs when contents of a Data Link Chain in a data block contains incorrect data. This error generated when data length in block exceeds 7FC.
016	RESIDUAL COUNT NOT ZERO AFTER REWRITE OPERATION (LARGE BUFFER).	On a rewrite all bytes are subtracted from the block length, the difference must equal zero. This error occurs when it is not.
017	INVALID UFB FIELD FOUND FOR REWRITE OPERATION (OFFSET=0).	User damaged segment 2 (UFB) data.
018	BUFFER POOL ERROR DETECTED. LOCKED BUFFER (BCE) IN CONTROL TABLE DOES NOT AGREE WITH CURRENT BUFFER (BCB).	User damaged segment 2 (UFB or BCT) data.
019	BLOCK TYPE (BCE) IN BUFFER POOL DOES NOT AGREE WITH CURRENT READ REQUEST.	Buffer Control Entry in the buffer pool is invalid.
020	BUFFER POOL ERROR DETECTED. BLOCK TYPE (BCE) IS INVALID FOR IO INITIATION.	Block Type in the buffer pool is invalid.
021	BUFFER POOL ERROR DETECTED. BUFFER (BCE) WITH IO IN PROGRESS NOT ON BCTBL CHAIN OR INTERNAL LOCK OTHER BCE OPERATION FAILED.	User damaged UFB or BCT.
022	ERROR DETECTED IN THE ALTERNATE INDEX DATA STRUCTURE, UNABLE TO LOCATE THE RECORD.	Alternate index block contains a primary key value which is not in any data block.
023	TRACE ROUTINE FOR DUPLICATE KEY VALUES FAILED. ALTERNATE TREE NOT MODIFIED.	The offset into the alternate index block is invalid for a user.
024	UNEXPECTED ERROR OCCURRED DURING FILE RESTORE OPERATION. WRITE OR REWRITE FUNCTION FOR ALTERNATE INDEX FILE FAILED DUE TO DUPLICATE KEY ERROR, AND ATTEMPT TO RESTORE FILE WAS UNSUCCESSFUL.	On a WRITE or REWRITE to an alternate-indexed file, if a duplicate key is encountered on a path with no duplicates allowed, the system attempts to delete the record from the primary tree and all alternate trees on which it has been written. This error occurs if the attempt fails unexpectedly. User should attempt COPY/REORG.

C.5 SVC CLOSE CANCEL MESSAGES

These messages refer to unexpected error conditions that rarely occur.

ERROR NUMBER	MESSAGE
E001	SVC CLOSE ISSUED FOR NON-OPENED FILE.
E002	DEALLOCATION ERR; OFB NOT FOUND.
E003	DEALLOCATION ERR; IORE QUEUED.
E004	CODE = ; UPDATFDR SVC ERR. NO DEALLOCATION.
E005	UNABLE TO DEALLOCATE BUFFER DUE TO INVALID BUFFER ADDRESS OR BUFFER LENGTH IN UFB.
E006	INVALID UFB POINTER RETURNED AFTER LAST DMS OPERATION.
E007	CODE = XX; SCRATCH SVC ERROR. FILE CLOSED OK, BUT NOT SCRATCHED.
E008	INVALID UFB ADDRESS PRESENTED TO SVC CLOSE.
E009	UNEXPECTED ERROR OCCURRED WHILE ATTEMPTING TO CLOSE FILE (SHARED MODE).
E010	UNABLE TO DEALLOCATE BUFFER WITHIN BUFFER POOL DUE TO INVALID ADDRESS OR LENGTH IN BUFFER CONTROL TABLE ENTRY.
E012	FAIL TO LOCATE PROGRAM BUILDALT. UNABLE TO BUILD ALTERNATE INDEXES.
E013	FILE LABEL NOT UPDATED ( ). USER PROGRAM HAS INCORRECTLY MODIFIED THE UFB.
E014	FILE LABEL NOT UPDATED ( ). VTOC ERROR DETECTED.
E015	SYSTEM ERROR - FREEHEAP FAILED.
E016	I/O ERROR WHILE UPDATING RECOVERY BLOCKS.
E017	BEFORE IMAGE JOURNAL READ OPERATION HAS FAILED.
E018	BEFORE IMAGE JOURNAL REWRITE OPERATION HAS FAILED.
E019	BEFORE IMAGE JOURNAL CLOSE FAILED.
E020	BEFORE IMAGE JOURNAL SCRATCH FAILED.
E021	UNEXPECTED ERROR WHILE UPDATING RECOVERY BLOCK.
E022	FREE ALL FAILURE WHILE CLOSING RECOVERED DMS/TX FILE
E023	SYSTEM ERROR - TASK DATABASE BLOCK (TDB) DEALLOCATION FAILED.
E025	NO BUFFER SPACE AVAILABLE TO UPDATE RECOVERY BLOCKS.

C.6 FILE STATUS (FS) CODES FOR DMS

DMS returns to the user program by means of the RETURN macroinstruction. Registers 2 through 15 are always restored. Register 0 (R0) is also restored unless UFBEODAD or UFBERRAD is used--R0 then contains the normal return address. Register 1 (R1) is also restored unless the Read-No-Data option has been used--R1 then contains the record address.

DMS indicates the result of the function request through file status bytes UFBFS1 and UFBFS2. These bytes generally contain a value of X'30' - X'39', corresponding to the ASCII characters 0 through 9, called the File Status (FS) Code. Within this manual file status codes are represented as character values. File Status Byte 1 (UFBFS1) indicates the general type of file status and File Status Byte 2 (UFBFS2) indicates a specific item within the group. The various UFBFS1 groups are defined as follows:

0 - Successful Completion
1 - End of File
2 - Record Not Found (Disk File)
3 - I/O Error or Boundary Violation
4 - ADMS Codes
6 - Cancel
7 - Time-Out
8 - Special Shared Mode Errors
9 - Miscellaneous -- This Group includes errors caused by incorrect user-supplied information; e.g., Invalid Function, Invalid Mask, Invalid Length, or Invalid Format.

Following is a list of File Status codes with a description of each code and the conditions under which it can occur.

FILE STATUS FOR NORMAL RETURNS

FS CODE	MEANING	FUNCTION REQUEST	DEVICE	FILE ORGANIZATION	MODE	CAUSE
00	Successful Completion.	N/A	Disk, Tape, Printer	N/A	N/A	N/A
0X	Successful Completion.	N/A	Workstation	N/A	N/A	UFBFS2 ('X' in code field) contains the AID byte.
02	Successful Completion.	Read	Disk	Alternate Indexed	N/A	After successfully complet-ing a READ KEYED or READ NEXT on an alternate key path, the return code is 02 indicating at least one more record exists with the same alternate key value.

FILE STATUS FOR UFBEODAD RETURN

FS CODE	MEANING	FUNCTION REQUEST	DEVICE	FILE ORGANIZATION	MODE	CAUSE
10	End of File Reached.	READ NEXT	Disk or Tape	N/A	Input, I/O, or Shared	End of file was reached.

FILE STATUS FOR UFBEODAD RETURN (cont'd)

FS CODE	MEANING	FUNCTION REQUEST	DEVICE	FILE ORGANIZATION	MODE	CAUSE
11	End of Volume.	READ NEXT	Tape	N/A	N/A	This code is returned if the user program indicates (by UFBTFLGEODEOV) that no automatic volume switch is desired.
21	Record Key Out of Sequence or Duplicate Key Found During Indexed File Creation.	WRITE	Disk	Indexed	Output	The current record key is not greater than the preceding record key.
22	Duplicate Key Value.	WRITE	Disk	Indexed	I/O or Shared	The record to be added to the file has the same key as an existing record in the file.
23	Record Not Found in File.	READ RELATIVE	Disk	Consecutive	Input or I/O	The supplied record number is equal to zero or greater than the highest record number in the file.
23	Record Not Found in File.	READ KEY or START (Equal)	Disk	Indexed	Input, I/O*, or Shared	There is no record in the file containing a key equal to the supplied key.
23	Record Fot Found in File.	READ or REWRITE	Disk	N/A	I/O	The supplied block number is beyond the end of the file.
24	Primary Extent Exceeded (Indexed File Creation).	WRITE	Disk	Indexed	Output	Primary extent exceeded. The record cannot be added to the file. The file may be closed successfully and then opened in I/O Mode to add more records.
24	Record Not Found. Key Supplied Greater Than Key Value in File.	START (Greater Than) or (Greater Than or Equal)	Disk	Indexed	Input, I/O, or Shared	The supplied key is greater than the highest key value in the file.

FILE STATUS FOR UFBERRAD RETURN

FS CODE	MEANING	FUNCTION REQUEST	DEVICE	FILE ORGANIZATION	MODE	CAUSE
30	Permanent IO Error IOSW = XXXXXXXX XXXXXXXX.	N/A	N/A	N/A	N/A	A physical I/O operation was attempted and a hardware error occurred. The error is logged separately by SVC CHECK. This file status is returned for hardware errors only; it is not returned for program related errors.
34	Workstation Order Check.	READ or REWRITE	Workstation	N/A	I/O	Invalid information supplied in the workstation order area; i.e., Invalid Cursor Position: Row 25 Column 10.
34	Boundary Violation (Extent Cannot Be Obtained).	WRITE	Disk	Consecutive	Output or Extend	There is no more space in the file for additional records. An additional extent is unavailable because either the maximum number of extents are already allocated or the extent size is not available on volume.
34	Boundary Violation (Extent Limit of 13 Has Been Reached).	WRITE	Disk	Indexed	I/O or Shared	There is no more space in the file for additional records (as above) due to extent limit (13) exceeded or no available extent on volume. For Shared mode, an additional extent may also be unavailable due to maximum number of additional extents per run already allocated.
60	DMS Cancel Condition Occurred; Cancel Message Suppressed.	N/A	Disk or Tape	N/A	Shared	User requested suppression of all DMS-Cancel messages. Process the file in non-Shared mode to set the error message flag. If a DMS error condition code with FS=60-019 occurs, refer to DMS error code 019 in Appendix C.
70	Shared Time-Out Condition.	N/A	N/A	N/A	N/A	This feature will be available with the Advanced Sharer.

FILE STATUS FOR UFBERRAD RETURN (cont'd)

FS CODE	MEANING	FUNCTION REQUEST	DEVICE	FILE ORGANIZATION	MODE	CAUSE
80	Invalid Key Area Found for Read Key or Start Key.	READ KEYED or START KEYED	N/A	Indexed	Shared	UFBKEYAREA does not point to the key embedded in the record; i.e., specifies the key has a value of one for a length of five but it actually has a value of two for a length of five.
81	Invalid Read No-Data Issued.	READ NO-DATA	Disk	Indexed or Consecutive	Shared	Attempting to do a Read No-Data in Shared mode which is an invalid function request.
82	Label Update Operation after Last Function Was Unsuccessful.	N/A	N/A	N/A	Shared	Internal error by DMS. The file label (FDR1) is updated whenever any of the following fields are modified by DMS: Root Block Number; First Data Block Number; or Count of Levels in the (Primary) Index. If UPDATFDR is unsuccessful, FS equals 82 is returned.
83	The Sharing Task Has Terminated and Must be Restarted.	N/A	N/A	N/A	Shared	Sharing task is functioning incorrectly. Must IPL the System to restart Sharer.
84	Invalid Record Size or Area Supplied for Shared Request.	N/A	N/A	N/A	Shared	User Attempted to rewrite a variable length record whose length is greater than the maximum record size specified in the VTOC.
85	Update Access Denied.	WRITE, REWRITE, or DELETE	N/A	N/A	Shared	User Attempted to update a file in Shared mode but has Read-Only access.
86	Resource Control Error.	N/A	N/A	N/A	Shared	Incorrect sequence of Shared function requests; e.g., Attempting to do a Start Hold on a file while another file is already held.
87	Deadlock error for DMS/TX files	READ HOLD or START HOLD	Disk	Indexed	Shared	Your task attempted to hold a requested resource and deadlocked with another task, preventing both tasks from proceeding. Upon expiration of the TIMEOUT, DMS performs transaction rollback, then issues this file status to each file in which resources were rolled back.

FILE STATUS FOR UFBERRAD RETURN (cont'd)

FS CODE	MEANING	FUNCTION REQUEST	DEVICE	FILE ORGANIZATION	MODE	CAUSE
95	Invalid Function Sequence.	REWRITE, DELETE, or READ NEXT HOLD	Disk	Indexed	IO or Shared	Invalid function sequence similar to consecutive file case above occurred. Also returned if Read Next Hold issued without a file block HELD (invalid sequence).
95	READ RELATIVE Invalid for Variable Length Records.	READ RELATIVE	Disk	Consecutive	Input, or IO	Read Relative is only valid for fixed-length consecutive files.
95	Invalid Function Request.	N/A	N/A	N/A	N/A	Valid function requests are described for the given combinations of device class, open mode and file organization supported by DMS. After a file has been opened, an invalid function request is flagged with FS equals 95. Example: attempting to write a record while the file is opened in Input mode.
95	Invalid Function Sequence.	REWRITE	Disk	N/A	Shared	Record was not read with the HOLD option. For Shared Mode, an intervening READ with HOLD on another file may have released the HOLD. A function sequence error exists since the record cannot be rewritten unless it is 'HELD'.
95	REWRITE Function Invalid for Consecutive File with Compressed Records.	REWRITE	Disk	Consecutive	IO	Consecutive files can be rewritten only for fixed-length records.
95	Invalid Function Issued on Alternate Indexed File.	N/A	N/A	N/A	N/A	N/A
95	READ NEXT Issued on Indexed File when Current Position Was Undefined.	N/A	N/A	N/A	N/A	N/A
95	Invalid Function Issued in Shared Mode.	N/A	N/A	N/A	N/A	N/A
95	Invalid START Function (Modifier Byte Error)	START	Disk	Consecutive or Indexed	Input, IO, or Shared	START function modifier byte does not correspond to a valid START option.
95	Primary Key Value Was Changed when Rewriting an Indexed Record.	REWRITE	Disk	Indexed or Alternate Indexed	IO or Shared	Attempted to change the value of the Primary Key while rewriting a record.

FILE STATUS FOR UFBERRAD RETURN (cont'd)

FS CODE	MEANING	FUNCTION REQUEST	DEVICE	FILE ORGANIZATION	MODE	CAUSE
96	Invalid Disk Address Detected.	N/A	N/A	N/A	N/A	Error usually not caused by user program. Error can occur for invalid disk address in the extent list (possibly caused by incorrect device arrangement at SYSGEN). This file status is returned only if the IOSW indicates invalid command or data address. Under RAM, DMS supplies the buffer area and command; thus, FS = 96 is a rare error under RAM.
96	Write Operation Attempted on Write-Protected Disk.	N/A	N/A	N/A	N/A	An attempt was made to write to an open write-protected diskette (this can occur if a user remounts a diskette changing it to write-protected but not using the MOUNT command).
96	Invalid Data Area Location or Alignment (IO Command error).	READ, REWRITE, or WRITE	N/A	N/A	Input, IO, or Output (Block Level)	Data area location is invalid or alignment is not on a page boundary. Data area location is checked with data area length to ensure that only the stack, static area, or buffer area is being used.
96	Same as 96 above.	READ or REWRITE	Workstation	N/A	IO	Invalid data area location or alignment (word alignment required.
97	Invalid Length when Rewriting Variable Length Record.	REWRITE	Disk	Indexed	N/A	Invalid length is indicated when attempting to rewrite a variable-length record whose length is longer than the value established in UFBRECSIZE.
97	Same as 97 above.	REWRITE	Disk	Consecutive	IO	Invalid, cannot change record length of a consecutive file.
97	Invalid Length Supplied when Writing Variable-Length Record.	WRITE	N/A		Output, Shared, IO, or Extend	Invalid length is indicated when attempting to write a variable-length record whose length is greater than the value established in UFBRECSIZE.

FILE STATUS FOR UFBERRAD RETURN (cont'd)

FS CODE	MEANING	FUNCTION REQUEST	DEVICE	FILE ORGANIZATION	MODE	CAUSE
97	Invalid Record-Prefix Found in Variable-Length Record.	N/A	N/A	N/A	N/A	Error encountered while DMS is attempting to extract a variable-length record from its buffer. Error should not normally be encountered by the user.
97	Invalid Length Specified for IO Operation.	N/A	Printer, Tape, or Workstation	N/A	N/A	Length specified is not valid for the device. For the printer, length is invalid if it equals zero or is larger than the length specified at SVC OPEN. For the workstation, length is invalid if data length and starting row cause screen overflow. For tape, length is invalid if a long block or a short block (with non-integral number of records) is read.
98	Invalid Alternate Tree Mask Supplied on Write or Re-write Function.	WRITE or REWRITE	Disk	Alternate Indexed	Output, IO, or Shared	Alternate key mask references a nonexistent alternate key. For Write or Re-write, the user-supplied mask must indicate valid ALT-trees and the alternate key fields must fall within the record; otherwise, FS=98 is returned. NOTE: A mask of zero is always valid.
99	Invalid Format Found for Current File Block.	N/A	N/A	N/A	N/A	A block within a variable-length record file has an invalid prefix, a VLEN record has an invalid prefix, or a compressed record has an invalid format when expanded.

APPENDIX D
DMS GETPARM SCREENS AND PROCEDURE LANGUAGE

D.1 INTRODUCTION TO GETPARMS

The VS Operating System supports a supervisor call, known as the GETPARM SVC, which solicits and accepts runtime parameter information, displays runtime messages and awaits their acknowledgement. GETPARM-generated prompts appear on the workstation screen during normal execution. These prompts solicit parameter information from a user or from a controlling procedure. The GETPARM SVC verifies values entered from either source for validity. If the values entered are not acceptable, the GETPARM SVC responds with an error message.

GETPARM processing differs from other methods of obtaining runtime information primarily because it can interface with a procedure (refer to the VS Procedure Language Reference for further information on coding Procedure Language). A procedure is the preferred source of information for a GETPARM request. Thus, GETPARM prompts never appear on the workstation screen when they are satisfied by a Procedure Language ENTER statement. When you use a Procedure Language DISPLAY statement, values supplied in the procedure are displayed on the workstation screen as modifiable defaults, overriding any defaults supplied in the program. The ENTER and DISPLAY Procedure Language instructions facilitate runtime specification of file definition parameters.

D.2 THE STRUCTURE OF A GETPARM

A parameter reference name (prname) identifies the program's GETPARM request for each file. The prname for each request is, in general, unique within that program.

Many GETPARM requests contain one or more modifiable fields into which a user or a procedure can enter information. A keyword identifies each of these fields. When a GETPARM request appears, the keyword displayed on the screen for each field provides a description of the information to be supplied for that field. Also, many GETPARM requests solicit a PF key response (such as 16 = Exit Program). No keyword is associated with a PF key choice; you specify only the PF key number itself.

D.3 ASSOCIATING A PROCEDURE WITH A GETPARM

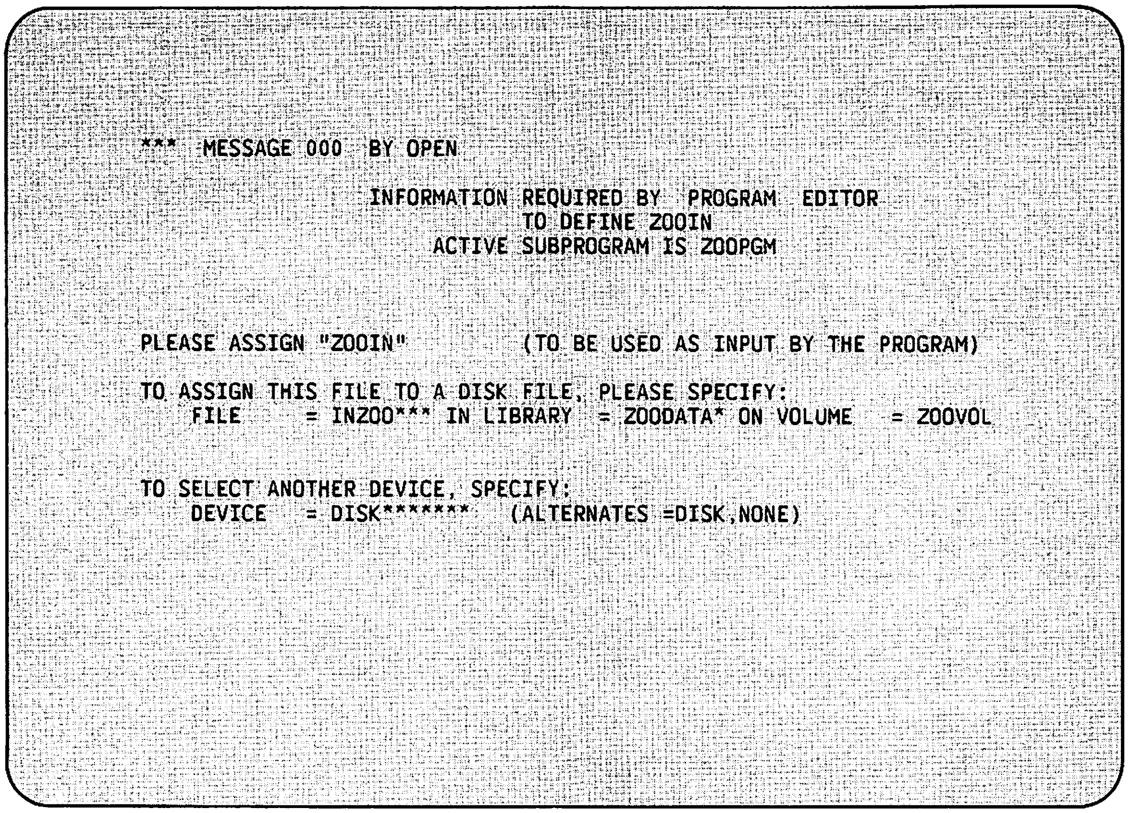
Within a procedure, each ENTER or DISPLAY statement supplies parameters for a single GETPARM request. The prname of a specific GETPARM request associates the request with a particular ENTER or DISPLAY statement. You can assign any prname to a GETPARM request. You specify the modifiable fields and their keywords for user-defined GETPARM requests.

When a procedure supplies parameters, keywords in the ENTER or DISPLAY statement associate the specified values with the fields to which they are to be assigned in the GETPARM request. In this case, the procedure passes the values associated with keywords to the corresponding keyword-identified fields in the GETPARM request. If the procedure does not assign new values to fields, they retain the default values supplied in the program or from the user defaults.

Refer to the VS Procedure Language Reference for more information on the Procedure Language and the use of GETPARM requests.

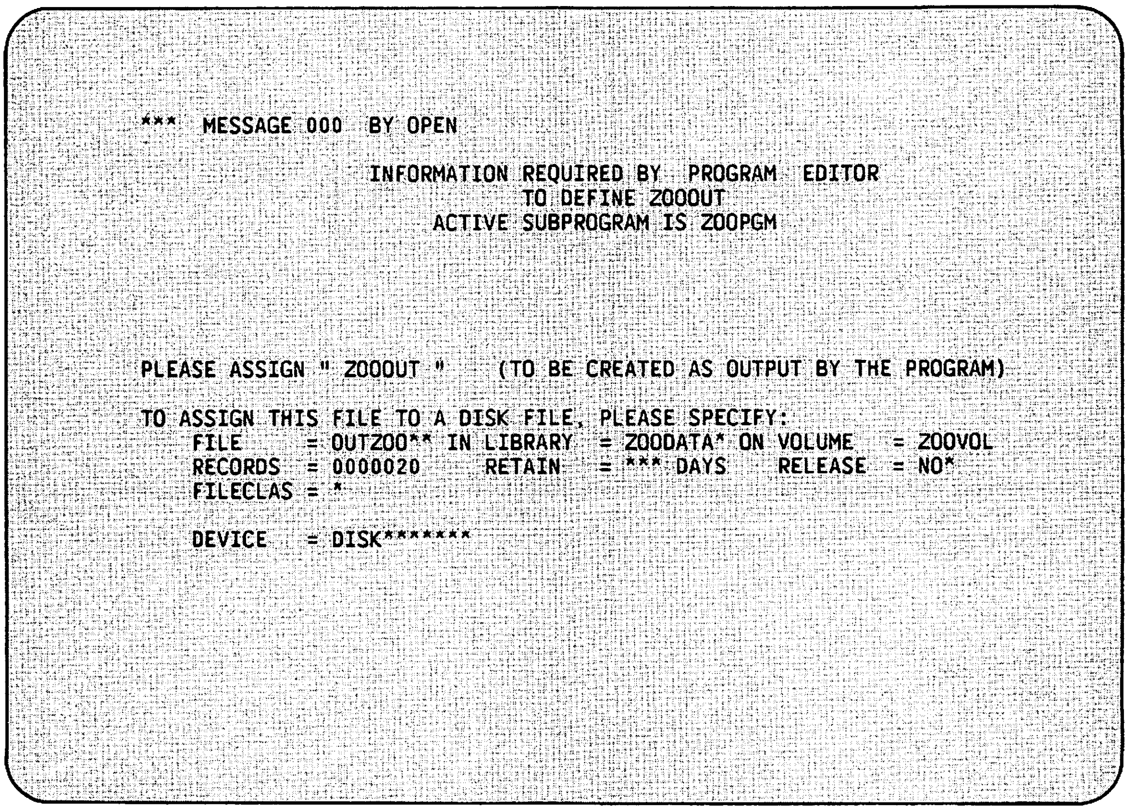
D.4 DMS FILE DEFINITION GETPARM SCREENS

DMS issues the following GETPARM screens to enable you to define DMS file parameters at runtime. Associated with each screen is a simple Procedure Language program to supply values to the modifiable fields of the screen.



```
*** MESSAGE 000  BY OPEN

                    INFORMATION REQUIRED BY  PROGRAM  EDITOR
                                TO DEFINE ZOOIN
                          ACTIVE SUBPROGRAM IS ZOOPGM


PLEASE ASSIGN "ZOOIN"         (TO BE USED AS INPUT BY THE PROGRAM)

TO ASSIGN THIS FILE TO A DISK FILE, PLEASE SPECIFY:
    FILE     = INZOO*** IN LIBRARY  = ZOODATA* ON VOLUME   = ZOOVOL


TO SELECT ANOTHER DEVICE, SPECIFY:
    DEVICE   = DISK*******   (ALTERNATES =DISK,NONE)
```

Figure D-1. GETPARM Screen for Input File Definition



```
*** MESSAGE 000  BY OPEN

                    INFORMATION REQUIRED BY  PROGRAM  EDITOR
                                TO DEFINE ZOOOUT
                          ACTIVE SUBPROGRAM IS ZOOPGM



PLEASE ASSIGN " ZOOOUT "    (TO BE CREATED AS OUTPUT BY THE PROGRAM)

TO ASSIGN THIS FILE TO A DISK FILE, PLEASE SPECIFY:
    FILE     = OUTZOO** IN LIBRARY  = ZOODATA* ON VOLUME   = ZOOVOL
    RECORDS  = 0000020     RETAIN   = *** DAYS    RELEASE  = NO*
    FILECLAS = *

    DEVICE   = DISK*******
```

Figure D-2. GETPARM Screen for Output File Definition

The procedure shown in Example D-1 supplies GETPARM file definition parameters to the screens shown in Figures D-1 and D-2 without displaying those screens on the workstation.

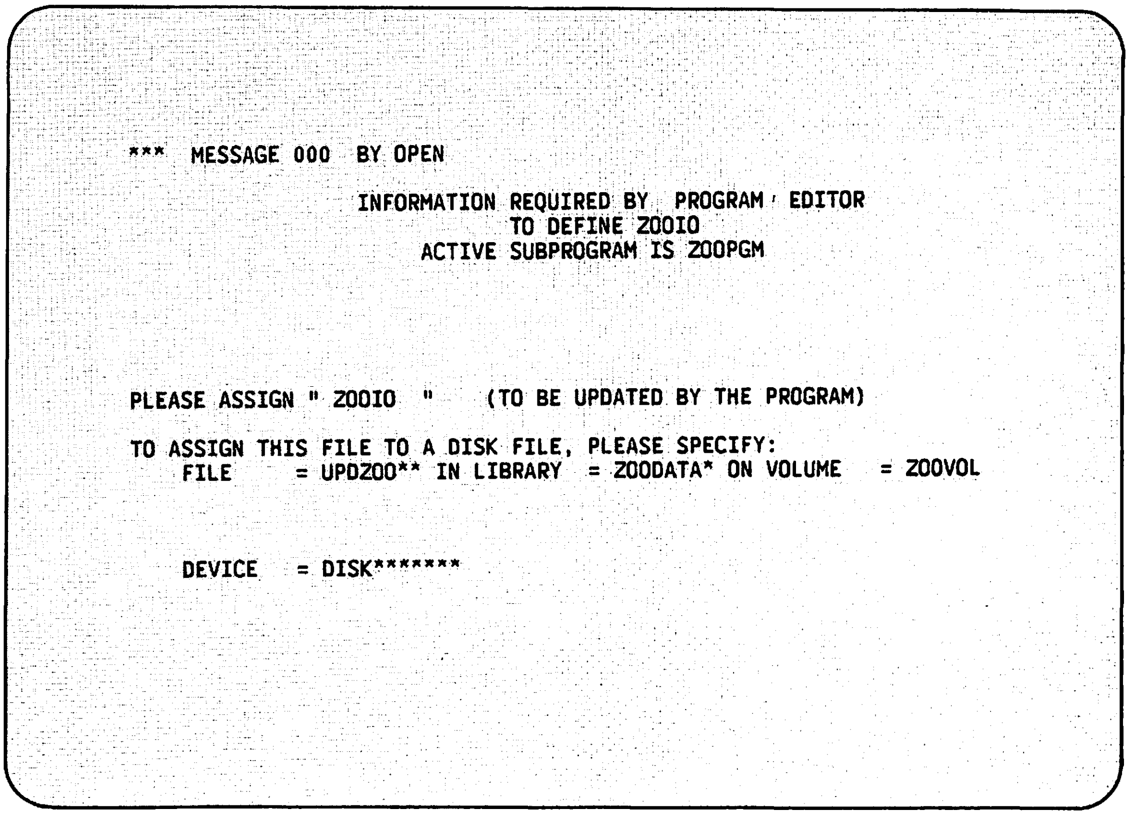
Example D-1. Procedure for Figures D-1 and D-2 (Non-display)

```
PROCEDURE
RUN ZOOPRG in ZOOLIB on ZOOVOL
ENTER ZOOOUT FILE = NEWZOO, LIBRARY = ZOODATA, VOLUME = ZOOVOL,
           RECORDS = 20
ENTER ZOOIN  FILE = INZOO, LIBRARY = ZOODATA, VOLUME = ZOOVOL
RETURN
```

The procedure shown in Example D-2 supplies GETPARM file definition parameters to the screens shown in Figures D-1 and D-2. The keyword values are displayed on the screens as modifiable defaults.

Example D-2. Procedure for Figures D-1 and D-2 (Display)

```
PROCEDURE
RUN ZOOPRG in ZOOLIB on ZOOVOL
DISPLAY ZOOOUT FILE = NEWZOO, LIBRARY = ZOODATA, VOLUME = ZOOVOL,
             RECORDS = 20
DISPLAY ZOOIN  FILE = INZOO, LIBRARY = ZOODATA, VOLUME = ZOOVOL
RETURN
```



```
***  MESSAGE 000  BY OPEN
                         INFORMATION REQUIRED BY  PROGRAM  EDITOR
                                     TO DEFINE ZOOIO
                              ACTIVE SUBPROGRAM IS ZOOPGM



PLEASE ASSIGN " ZOOIO  "    (TO BE UPDATED BY THE PROGRAM)

TO ASSIGN THIS FILE TO A DISK FILE, PLEASE SPECIFY:
    FILE      = UPDZOO** IN LIBRARY  = ZOODATA* ON VOLUME   = ZOOVOL


    DEVICE    = DISK*******
```

Figure D-3. GETPARM Screen for Update File Definition

The procedure shown in Example D-3 supplies GETPARM file definition parameters to the screen shown in Figure D-3. The keyword values are displayed on the screens as modifiable defaults. You can write this procedure with either a DISPLAY or ENTER statement.

Example D-3. Procedure for Figure D-3

```
PROCEDURE
RUN ZOOPRG in ZOOLIB on ZOOVOL
DISPLAY ZOOIO FILE = UPDZOO, LIBRARY = ZOODATA, VOLUME = ZOOVOL
RETURN
```

APPENDIX E
SAMPLE ASSEMBLY LANGUAGE PROGRAMS

E.1 RAM ALTERNATE INDEXED FILE UPDATE PROGRAM

```
************************************************************************
*                                                                      *
*      Demonstration program - Alternate-indexed file manipulations    *
*      The places in the program where the file is closed and then     *
*      re-opened are convenient points to set breakpoint traps in      *
*      order to cancel the program and inspect the file via DISPLAY.   *
*                                                                      *
*      Programmer:  G. Morrow                                          *
*            Date:  10/83                                              *
*                                                                      *
************************************************************************
         PRINT NOGEN
*
* Program equates
*
         REGS                           System register equate macro
RUFB     EQU   R11                      R11 will address ZUFB
RAXD1    EQU   R10                      R10 will address AXD1
RWORK    EQU   R9                       R9 is work register
RALTKEY  EQU   R8                       R8 holds alternate key
RPKEY    EQU   R7                       R7 holds primary key
*
* Set up basic addressability.
*
ALTDEMO  CODE
         BALR  EP,0                     Address this code
         USING *,EP                     Tell the assembler
         LR    R12,R14                  (R12) = static base pointer
         AL    R12,=R(ALTSTAT)          Add offset of program static
         USING ALTSTAT,R12              Tell assembler where static is
         LA    RUFB,ALTFILE             Address the data file UFB
         USING UFB,RUFB                 UFB references use R11
         USING AXD1,RAXD1               AXD1 references use R10
*
         OPEN  UFB=ALTFILE,MODE=OUTPUT  Open the data file for output
         OPEN  UFB=WSFILE,MODE=IO       Open the workstation
         LA    RAXD1,AXD                Address the AXD1 block
*
```

```
         LA    RPKEY,1                 Initialize primary key reg
         LA    RALTKEY,2100            Ditto alternate key reg
         LA    RWORK,100               Use work reg as loop counter
         MVC   SCRNMSG(9),=C' Writing '  Header to screen
*
* Write 100 records - PK: 1 thru 100;  AK1: 1001 thru 1100;
* AK2: 2100 thru 2001 by -1.  Enable both alternate paths.
*
WRTLOOP  CVD   RPKEY,PACKED            Convert primary key to decimal
         UNPU  PKEY(4),PACKED(8)       Then to display format in record
         MVC   ALTKEY1,PKEY            Alternate key #1 = primary key
         MVI   ALTKEY1,C'1'            Add 1000 to alt key #1
         CVD   RALTKEY,PACKED          Convert alt key #2 to decimal
         UNPU  ALTKEY2(4),PACKED(8)    Then to display format in record
         MVC   AXD1MASK(2),BOTH        Set mask for both alt paths
         JSI   =A(DISP)                Display the record as set up
         WRITE UFB=ALTFILE             Write record to disk
         LA    RPKEY,1(,RPKEY)         Increment primary key reg
         BCTR  RALTKEY,0               Decrement alternate key reg
         BCT   RWORK,WRTLOOP           Loop until 100 records written
*
* Close the file and re-open it in IO mode.
*
         CLOSE UFB=ALTFILE             Close the data file
         OPEN  UFB=ALTFILE,MODE=IO     Re-open the data file
*
* Address the AXD1 block.
*
         L     RAXD1,UFBALTPTR         Load AXD1 pointer from UFB
         LA    RAXD1,0(,RAXD1)         Clear the high byte
*
* Set up to read the file sequentially along alternate path #1.
*
         MVC   ALTKEY1(4),=XL4'0'      Set alt key #1 to binary zeroes
         MVI   AXD1ALTINX,1            Set key of reference to path #1
         LA    RWORK,ALTKEY1           Get address of alternate key #1
         ST    RWORK,UFBKEYAREA        Set UFB key pointer to it
         START GE,UFB=ALTFILE          Start at 1st rec on alt path #1
         LA    RWORK,RWRTEND           Get end-of-data address
         ST    RWORK,UFBEODAD          Store in UFB
         MVC   SCRNMSG(9),=C'Rewriting' Header to screen
*
* Read file sequentially along Alternate Path #1 in pairs of records.
* For the first record of each pair, remove the record from the second
* alternate path.  For the second record of each pair, remove the
* record from the first alternate path.  When done, the records with
* odd primary keys are enabled on Alternate Path #1 and the records
* with even primary keys are enabled on Alternate Path #2.
*
RWRTLOOP READ  HOLD,UFB=ALTFILE        Read first record of pair
         MVC   AXD1MASK(2),PATH1       Enable only 1st alt path
         JSI   =A(DISP)                Display the record
         REWRITE UFB=ALTFILE           Rewrite the record
```

```
*
         READ  HOLD,UFB=ALTFILE         Read second record of pair
         MVC   AXD1MASK(2),PATH2        Enable only 2nd alt path
         JSI   =A(DISP)                 Display the record
         REWRITE UFB=ALTFILE            Rewrite the record
         B     RWRTLOOP                 Loop until EOD exit taken
*
* Close the file and re-open it in IO mode.
*
RWRTEND  CLOSE UFB=ALTFILE              Close the data file
         OPEN  UFB=ALTFILE,MODE=IO      Re-open the data file
*
* Address the AXD1 block.
*
         L     RAXD1,UFBALTPTR          Load AXD1 pointer reg from UFB
         LA    RAXD1,0(,RAXD1)          Clear the high byte
*
* Set up to read the file sequentially along Alternate Path #2.  (This
* means we will be reading in reverse primary key order because of the
* way the records were originally set up.)
*
         MVC   ALTKEY2(4),=XL4'0'       Set alt key #2 to binary zeroes
         MVI   AXD1ALTINX,2             Set key of reference to path 2
         LA    RWORK,ALTKEY2            Get address of alt key #2
         ST    RWORK,UFBKEYAREA         Set UFB key pointer to it
         START GE,UFB=ALTFILE           Position to first record on path
         LA    RWORK,DELETEND           Get addr of end-of-data routine
         ST    RWORK,UFBEODAD           Set it down in UFB
         MVC   SCRNMSG(9),=C'Deleting ' Header to screen
*
* Read file sequentially along Alternate Path #2 and delete every
* other record encountered from the file.  We should wind up with
* 75 records in the file, with 50 of them enabled on the first
* alternate path and 25 enabled on the second alternate path.
*
DELTLOOP READ  HOLD,UFB=ALTFILE         Read a record
         JSI   =A(DISP)                 Display the record
         DELETE UFB=ALTFILE             Delete the record
         READ  HOLD,UFB=ALTFILE         Read a record
         JSI   =A(DISP)                 Display the record (no delete)
         B     DELTLOOP                 Loop until EOD exit taken
*
* Close the data file and the workstation and end the program.
*
DELETEND CLOSE UFB=ALTFILE              Close the data file
         CLOSE UFB=WSFILE               Close the workstation
         RETURN UNLINK                  Exit the program
*
* Record display subroutine.
*
DISP     MVC   L10+4(18),PKEY-14        Primary key + text to screen
         MVC   L10+27(23),ALTKEY1-19    Alt key #1 + text to screen
         MVC   L10+55(23),ALTKEY2-19    Alt key #2 + text to screen
         MVC   ORDAREA,=XL4'01000000'   Set up order area
         REWRITE UFB=WSFILE             Rewrite the screen
         RT                             Return to caller
*
         LTORG
*
ALTSTAT  STATIC
RECORD   DS    0F
         DC    C'Primary key = '
PKEY     DS    CL4
         DC    CL14' '
         DC    C'Alternate key #1 = '
ALTKEY1  DC    CL4'0000'
         DC    CL9' '
         DC    C'Alternate key #2 = '
ALTKEY2  DS    CL4
         DC    CL9' '
*
ALTFILE  UFBGEN PRNAME=ALTINX,FILENAME=ALTDEMO,LIBRARY=RWLDATA,       X
               VOLSER=WORK,DEVCLASS=DISK,FORG=INDEXED,RECSIZE=96,     X
               NRECS=100,KPOS=14,KSIZE=4,ALTCNT=2,ALTAREA=AXD,        X
               RECAREA=RECORD,KEYAREA=PKEY,NODISPLAY=YES
*
AXD      AXDGEN ENTRIES=2,                                            X
               (ORD=1,KEYPOS=51,KEYSIZE=4),                           X
               (ORD=2,KEYPOS=83,KEYSIZE=4,NODUPS)
*
BOTH     DC    X'C000'                  Path mask (paths 1 & 2)
PATH1    DC    X'8000'                  Path mask (path 1 only)
PATH2    DC    X'4000'                  Path mask (path 2 only)
NEITHER  DC    X'0000'                  Path mask (neither path)
PACKED   DS    D      Work area for external format conversion
*
WSFILE   UFBGEN PRNAME=CRT,DEVCLASS=WS,RECSIZE=1924,RECAREA=SCREEN
*
SCREEN   DS    0F                       Workstation screen definition
ORDAREA  DS    XL4                      Workstation order area
L1#6     DC    6CL80' '                 Lines 1 - 6 are blank
L7       DC    CL35' '                  Header msg leading Blanks
         DC    X'84'                    FAC (bright protect all noline)
SCRNMSG  DC    CL44' '                  Header msg goes here
L8#9     DC    2CL80' '                 Lines 8 & 9 are blank
L10      DC    CL80' '                  Key display line
L11#24   DC    14CL80' '                Lines 11 - 24 are blank
*
         UFB                            Declare UFB to assembler
         AXD1                           Declare AXD1 to assembler
         END
```

E.2 RAM WORKSTATION FILE DATA ENTRY PROGRAM

```
***********************************************************************
*                                                                     *
*       Simple data entry program which demonstrates reading          *
*       data from the workstation and writing it to a                 *
*       consecutive disk file.                                        *
*                                                                     *
*       Programmer:  G. Morrow                                        *
*            Date:  10/83                                             *
*                                                                     *
***********************************************************************
         PRINT NOGEN
*
*        Program equates
*
         REGS                       Register equate macro
RUFB     EQU   R4                   R4 used to address UFBs
HIDE     EQU   X'9C'                "Blank protect all noline" FAC
BLINK    EQU   X'94'                "Blink protect all noline" FAC
BEEP     EQU   X'E0'                "Unlock, beep & position cursor" WCC
NOBEEP   EQU   X'A0'                "Unlock & position cursor" WCC
*
DEMOCODE CODE
*
* Set up program and data addressability.
*
         BALR  EP,0                 Address this code section
         USING *,EP                 Tell the assembler
         LR    R12,R14              (R12) = Static section base ptr
*                                           for this link level
         AL    R12,=R(DEMOSTAT)     Add offset of program static section
         USING DEMOSTAT,R12         Static section now addressed
         USING UFB,RUFB             Tell assembler that R4 will be the
*                                        base reg for UFB addressing
*
* Open disk and workstation files, do some initialization.
*
         OPEN  UFB=WSUFB,MODE=IO    Open the workstation
         OPEN  UFB=DISKUFB,MODE=OUTPUT  Open the disk file
         LA    RUFB,DISKUFB         Temporarily address disk UFB
         OI    UFBF4,UFBF4RLSE      Turn on "release excess space" bit
         LA    RUFB,WSUFB           Permanently address workstation UFB
         MVI   MSGFAC,HIDE          Turn off error message line
         MVI   WCC,NOBEEP           Turn off workstation beep
*
* Get the input from the workstation.
*
DISPLAY  JSI   =A(INITSCRN)         Initialize screen fields
         REWRITE UFB=WSUFB          Display data entry screen
         READ  MOD,UFB=WSUFB        Read the workstation
```

```
*
* Process the input:  If PF 16 was struck, then quit.
*                     If ENTER was struck, then write data to the disk
*                                file and re-initialize the screen
*                     Otherwise re-initialize the screen and display
*                                the error message
*
         CLI   UFBFS2,C'P'         PF 16 struck ?
         BE    EXIT                Yes, quit
         CLI   UFBFS2,C'@'         ENTER key struck ?
         BE    TRANSFER            Yes, transfer data to disk file
*
* An invalid key was struck.
*
         MVI   WCC,BEEP            Turn on workstation beep bit
         MVI   MSGFAC,BLINK        "Unhide" the error message
         B     DISPLAY             Return to re-display screen
*
*
* The ENTER key was struck.
*
TRANSFER MVC   DATANAME,SCRNNAME   Move name field to disk record
         MVC   DATAADDR,SCRNADDR   Move address field to disk record
         MVC   DATAAREA,SCRNAREA   Move area code field to disk record
         MVC   DATAPRFX,SCRNPRFX   Move phone # prefix to disk record
         MVC   DATASUFX,SCRNSUFX   Move phone # suffix to disk record
         WRITE UFB=DISKUFB         Write the data record to disk
         MVI   MSGFAC,HIDE         Blank out error msg via it's FAC
         MVI   WCC,NOBEEP          Turn off workstation beep
         B     DISPLAY             Return to re-display screen
*
* PF 16 was struck.
*
EXIT     CLOSE UFB=DISKUFB         Close the data file
         CLOSE UFB=WSUFB           Close the workstation
         RETURN UNLINK             Exit the program
*
* Mapping area re-initialization subroutine.
*
INITSCRN MVPC  SCRNNAME(20),*+2(1),X'0B' Pseudo-blanks to name
         MVPC  SCRNADDR(25),*+2(1),X'0B' Pseudo-blanks to address
         MVPC  SCRNAREA(3),*+2(1),X'0B'  Pseudo-blanks to area code
         MVPC  SCRNPRFX(3),*+2(1),X'0B'  Pseudo-blanks to phone prefix
         MVPC  SCRNSUFX(4),*+2(1),X'0B'  Pseudo-blanks to phone suffix
         MVI   CURCOL,29                 Set cursor column
         MVI   CURROW,8                  Set cursor row
         RT                              Done!
*
         LTORG
*
DEMOSTAT STATIC
DISKUFB  UFBGEN PRNAME=DATA,DEVCLASS=DISK,FORG=CONSEC,RECSIZE=59,       &
               NRECS=50,RECAREA=DISKREC,NODISPLAY=YES,FILENAME=PHONES, &
               LIBRARY=RWLDATA,VOLSER=WORK
```

```
*
DISKREC  DS    0CL59                Disk file record defined here
DATANAME DS    CL20                 Name
DATAADDR DS    CL25                 Address
DATAPHON DC    C'('                 Constant (left paren)
DATAAREA DS    CL3                  Area code
         DC    C') '                Constant (right paren)
DATAPRFX DS    CL3                  Phone prefix
         DC    C'-'                 Constant (dash)
DATASUFX DS    CL4                  Phone suffix
*
WSUFB    UFBGEN PRNAME=CRT,DEVCLASS=WS,RECSIZE=1924,RECAREA=SCREEN
*
         DS    0F                   Force word alignment
SCREEN   DS    0CL1924              Worstation screen defined here
ORDAREA  DS    0XL4                 Order area defined here
MAPSTART DC    X'01'                Start screen rewrite from line 1
WCC      DS    X                    Write Control Character
CURCOL   DS    X                    Cursor column position
CURROW   DS    X                    Cursor row position
*
MAPAREA  DS    0CL1920              Mapping area defined here
LINES1#3 DC    3CL80' '             3 Blank lines at top of screen
*
LINE4    DC    CL31' '              Leading spaces
HDRFAC   DC    X'AC'                "Dim protect all underline" FAC
HDR      DC    C'Data Entry Demo'   Header message
         DC    X'8C'                "Dim protect all noline" FAC
         DC    CL32' '              Trailing spaces
*
LINES5#7 DC    3CL80' '             Lines 5 thru 7 are blank
*
LINE8    DC    CL22' '              Leading spaces
         DC    C'Name:'             Name field prompt
         DC    X'80'                "Bright modify all noline" FAC
SCRNNAME DS    CL20                 Modifiable name field
         DC    X'8C'                "Dim protect all noline" FAC
         DC    CL31' '              Trailing spaces
*
LINE9    DC    CL19' '              Leading spaces
         DC    C'Address:'          Address prompt
         DC    X'80'                "Bright modify all noline" FAC
SCRNADDR DS    CL25                 Modifiable address field
         DC    X'8C'                "Dim protect all noline" FAC
         DC    CL26' '              Trailing spaces
*
LINE10   DC    CL21' '              Leading spaces
         DC    C'Phone: ('          Phone # prompt
         DC    X'80'                "Bright modify all noline" FAC
SCRNAREA DS    CL3                  Modifiable area code field
         DC    X'8C'                "Dim protect all noline" FAC
         DC    C')'                 Area code right paren
         DC    X'80'                "Bright modify all noline" FAC
```

```
SCRNPRFX DS    CL3                     Modifiable phone # prefix
         DC    X'80'                   "Bright modify all noline" FAC
SCRNSUFX DS    CL4                     Modifiable phone # suffix
         DC    X'8C'                   "Dim protect all noline" FAC
         DC    CL35' '                 Trailing spaces
*
LINES11#14 DC  4CL80' '                Lines 11 - 14 are blank
*
LINE15   DC    CL14' '                 Leading spaces
         DC    C'Press <ENTER> to add data to disk file, PF16 to exit'
         DC    CL14' '                 Trailing spaces
*
LINES16#19 DC  4CL80' '                Lines 16 - 19 are blank
*
LINE20   DC    CL21' '                 Leading spaces
MSGFAC   DS    X                       Error message FAC
         DC    C'Invalid PFKEY - Data not transferred'
         DC    CL22' '                 Trailing spaces
*
LINES21#24 DC  4CL80' '                Lines 21 - 24 are blank
*
         UFB                           Make UFB DSECT known to assembler
         END
```

E.3 BAM FILE COPY PROGRAM

```
***********************************************************************
*                                                                     *
*       Demonstration program - BAM file copy                         *
*                                                                     *
*       Programmer:  G. Morrow                                        *
*            Date:  10/83                                             *
*                                                                     *
***********************************************************************
         PRINT NOGEN
         REGS
*
* Set up basic adressability.
*
BAMCOPY  CODE
         BALR  EP,0                Address this code
         USING *,EP                Tell the assembler
         LR    R12,R14             (R12) = static base pointer
         AL    R12,=R(BAMSTAT)     Add offset of program static section
         USING BAMSTAT,R12         Tell the assembler
*
* Use GETPARM to get the input and output file names.
*
         GETPARM KEYLIST=BAMKEYL,MSG=BAMMSG,FORM=REQUEST
*
* Open the input file.
*
         MVC   UFBIFILENAME,BINFILE+12 File name to UFB
         MVC   UFBIDIRNAME,BINLIB+12   Library name to UFB
         MVC   UFBIVOLSER,BINVOL+12    Volume name to UFB
         OPEN  UFB=INUFB,MODE=INPUT    Open the input file
*
* Copy file definition parameters from input file UFB to output file
* UFB and also move in the file, library, and volume from the GETPARM.
*
         MVC   UFBOFILENAME,BOUTFILE+12 File name to UFB
         MVC   UFBODIRNAME,BOUTLIB+12   Library name to UFB
         MVC   UFBOVOLSER,BOUTVOL+12    Volume name to UFB
         MVC   UFBOFORG,UFBIFORG        File organization to UFB
         XR    R4,R4                    Clear R4 to zeroes
         ICM   R4,B'0111',UFBIEBLK      (R4) = input file EBLK
         LA    R4,1(,R4)                Add 1 for blocks used
         STCM  R4,B'0111',UFBONBLKS     Store in output file NBLKS
         MVC   UFBORECSIZE,UFBILRECSAVE Logical record size to UFB
         MVC   UFBOFLAGS,UFBIFLAGS      Flags byte to UFB
```

```
*
* Move alternate index count and indexed file state fields from input
* UFB to output UFB in case the file is indexed.  Then, open the
* output file.  Set the output file record area to be the input file's
* DMS buffer.  This way, we can use NODATA on reads from the input
* file, thus saving the overhead of data movement to 2 separate
* record areas.
*
         MVC   UFBOALTCNT,UFBIALTCNT    Alternate index count to UFB
         MVC   UFBOKEYPOS(UFBINXDISKEND-UFBDMSEND),UFBIKEYPOS
*                                   Indexed state fields to UFB
         OPEN  UFB=OUTUFB,MODE=OUTPUT   Open the output file
         MVC   UFBORECAREA,UFBIBUFADR   Set output file record area
*
* This is the main read/write loop.  The EOD address is set in the
* input file UFB so that control will transfer to ENDCOPY when the
* input file has been completely .read.
*
COPYLOOP READ  NODATA,UFB=INUFB      Read a block from the input file
         WRITE UFB=OUTUFB            Write the block to the output file
         B     COPYLOOP              Branch back to copy next block
*
* All blocks copied.  Set the final file state fields in the output
* file's UFB to be written to the VTOC and close both files.
*
ENDCOPY  MVC   UFBONRECS,UFBINRECS   # of records to UFB
         MVC   UFBOEBLK,UFBIEBLK     Last block used to UFB
         MVC   UFBOEREC,UFBIEREC     EREC to UFB
         CLOSE UFB=UFBIBEGIN         Close input file
         CLOSE UFB=UFBOBEGIN         Close output file
         RETURN UNLINK               Exit the program
*
         LTORG
*
BAMSTAT  STATIC
*
* Input file UFB definition.  Supply the bare minimum so that any
* disk file can be opened.
*
         UFB   NODSECT,SUFFIX=I
         ORG   UFBIBEGIN
INUFB    UFBGEN PRNAME=IN,DEVCLASS=DISK,BAM=YES,NODISPLAY=YES,         &
               EODAD=ENDCOPY
*
* Output file UFB definition.  Same as above, except that the block
* allocation bit is set and no end-of-data address is needed.
*
         UFB   NODSECT,SUFFIX=O
         ORG   UFBOBEGIN
OUTUFB   UFBGEN PRNAME=OUT,DEVCLASS=DISK,BAM=YES,NODISPLAY=YES,        &
               BLKAL=YES
```

```
*
* GETPARM keylist and message list definitions.
*
BAMKEYL  KEYLIST PRNAME='FILES',LABELPFX='B',                          &
               'INFILE',('        ',UCHAR,2),                          &
               'INLIB',('        ',UCHAR),                             &
               'INVOL',('      ',UCHAR),                               &
               'OUTFILE',('        ',UCHAR,2),                         &
               'OUTLIB',('        ',UCHAR),                            &
               'OUTVOL',('      ',UCHAR)
*
BAMMSG   MSGLIST '000','DEMO',                                         &
               'Please identify the input and output files'
*
         UFB             Make the UFB DSECT known to the assembler
         END
```

INDEX

WANG

Customer Comment Form

Publication Number 800-1124-01

Title VS DATA MANAGEMENT SYSTEM REFERENCE

Help Us Help You . . .

We've worked hard to make this document useful, readable, and technically accurate. Did we succeed? Only you can tell us! Your comments and suggestions will help us improve our technical communications. Please take a few minutes to let us know how you feel.

How did you receive this publication?

- ☐ Support or Sales Rep
- ☐ Wang Supplies Division
- ☐ From another user
- ☐ Enclosed with equipment
- ☐ Don't know
- ☐ Other ______

How did you use this Publication?

- ☐ Introduction to the subject
- ☐ Classroom text (student)
- ☐ Classroom text (teacher)
- ☐ Self-study text
- ☐ Aid to advanced knowledge
- ☐ Guide to operating instructions
- ☐ As a reference manual
- ☐ Other ______

Please rate the quality of this publication in each of the following areas.

	EXCELLENT	GOOD	FAIR	POOR	VERY POOR
Technical Accuracy — Does the system work the way the manual says it does?	☐	☐	☐	☐	☐
Readability — Is the manual easy to read and understand?	☐	☐	☐	☐	☐
Clarity — Are the instructions easy to follow?	☐	☐	☐	☐	☐
Examples — Were they helpful, realistic? Were there enough of them?	☐	☐	☐	☐	☐
Organization — Was it logical? Was it easy to find what you needed to know?	☐	☐	☐	☐	☐
Illustrations — Were they clear and useful?	☐	☐	☐	☐	☐
Physical Attractiveness — What did you think of the printing, binding, etc?	☐	☐	☐	☐	☐

Were there any terms or concepts that were not defined properly? ☐ Y ☐ N If so, what were they? ______

After reading this document do you feel that you will be able to operate the equipment/software? ☐ Yes ☐ No ☐ Yes, with practice

What errors or faults did you find in the manual? (Please include page numbers) ______

Do you have any other comments or suggestions? ______

Name ______ **Street** ______

Title ______ **City** ______

Dept/Mail Stop ______ **State/Country** ______

Company ______ **Zip Code** ______ **Telephone** ______

Thank you for your help.

WANG

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 16 LOWELL, MA

POSTAGE WILL BE PAID BY ADDRESSEE

WANG LABORATORIES, INC.
TECHNICAL PUBLICATIONS
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851

Fold

Cut along dotted line.

WANG

To Order by Phone, Call:
(800)225-0234

Order Form for Wang Manuals and Documentation

① Customer Number (If Known)

② Bill To: ______________________ Ship To: ______________________

③ Customer Contact: ( ) ( ) Phone Name ④ Date Purchase Order Number

⑤ Taxable Yes ☐ No ☐ ⑥ Tax Exempt Number ⑦ Credit This Order to A Wang Salesperson Please Complete Salesperson's Name Employee No. RDB No.

⑧ Document Number	Description	Quantity	⑨ Unit Price	Total Price

⑩ Authorized Signature Date

☐ Check this box if you would like a free copy of the **Corporate Publications Literature Catalog** (700-5294)

Sub Total	
Less Any Applicable Discount	
Sub Total	
LocalStateTax	
Total Amount	

Ordering Instructions

1. If you have purchased supplies from Wang before, and know your Customer Number, please write it here.
2. Provide appropriate Billing Address and Shipping Address.
3. Please provide a phone number and name, should it be necessary for WANG to contact you about your order.
4. Your purchase order number and date.
5. Show whether order is taxable or not.
6. If tax exempt, please provide your exemption number.
7. If you wish credit for this order to be given to a WANG salesperson, please complete
8. Show part numbers, description and quantity for each product ordered.
9. *Pricing extensions and totaling can be completed at your option; Wang will refigure these prices and add freight on your invoice.*
10. Signature of authorized buyer and date.

Wang Supplies Division Terms and Conditions

1. **TAXES** — Prices are exclusive of all sales, use, and like taxes.
2. **DELIVERY** — Delivery will be F.O.B. Wang's plant. Customer will be billed for freight charges; and unless customer specifies otherwise, all shipments will go best way surface as determined by Wang. Wang shall not assume any liability in connection with the shipment nor shall the carrier be construed to be an agent of Wang. If the customer requests that Wang arrange for insurance the customer will be billed for the insurance charges.
3. **PAYMENT** — Terms are net 30 days from date of invoice. Unless otherwise stated by customer, partial shipments will generate partial invoices.
4. **PRICES** — The prices shown are subject to change without notice. Individual document prices may be found in the Corporate Publications Literature Catalog (700-5294)
5. **LIMITATION OF LIABILITY** — In no event shall Wang be liable for loss of data or for special, incidental or consequential damages in connection with or arising out of the use of or information contained in any manuals or documentation furnished hereunder.

WANG

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 16 NO. CHELMSFORD, MA.

POSTAGE WILL BE PAID BY ADDRESSEE

WangDirect
Attention: Order Entry Dept.
800 Chelmsford Street
Lowell, MA 01851

Fold

WANG

ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851
TEL. (617) 459-5000
TWX 710-343-6769, TELEX 94-7421

Printed in U.S.A.
800-1124-01
1-84-5M