NonStop™ Systems
NonStop 1+™ System

ENSCRIBE™ Programming Manual

Data Management Library

82583

NOTICE

Effective with the B00/E08 software release, Tandem introduced a more formal nomenclature for its software and systems.

The term "NonStop 1+™ system" refers to the combination of NonStop 1+ processors with all software that runs on them.

The term "NonStop™ systems" refers to the combination of NonStop II™ processors, NonStop TXP™ processors, or a mixture of the two, with all software that runs on them.

Some software manuals pertain to the NonStop 1+ system only, others pertain to the NonStop systems only, and still others pertain both to the NonStop 1+ system and to the NonStop systems.

The cover and title page of each manual clearly indicate the system (or systems) to which the contents of the manual pertain.

NonStop™ Systems
NonStop 1+™ System

ENSCRIBE™
Programming Manual

Abstract
This manual describes each of the four types of disc files supported by ENSCRIBE and how to access, create, and fill those files.

Product Version
ENSCRIBE B00 (NonStop Systems)
ENSCRIBE E08 (NonStop 1+ System)

Operating System Version
GUARDIAN B00 (NonStop Systems)
GUARDIAN E08 (NonStop 1+ System)

Part No. 82583 A00

March 1985

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014-2599

DOCUMENT HISTORY

Edition	Part Number	Operating System Version	Date
First Edition	82083 A00	GUARDIAN A00/E00	April 1981
Second Edition	82083 B00	GUARDIAN A05/E06	April 1983
Third Edition	82583 A00	GUARDIAN B00/E08	March 1985

New editions incorporate all updates issued since the previous edition.

82583 A00 3/85

NEW AND CHANGED INFORMATION

This is the third edition of the ENSCRIBE Programming Manual.

Technical and typographical errors have been corrected and product changes for the current release have been incorporated, primarily the effects of the new, optional DP2 disc process.

Also, much redundant material has been removed. The procedure-call syntax descriptions, which were the bulk of Section 3 in the previous editions, have been included in the new System Procedure Calls Reference Manual. Also, much of Section 1 has been removed because it is included in the GUARDIAN Operating System Programmer's Guide.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This manual documents ENSCRIBE, Tandem's data-base record manager. It is written for programmers and administrators whose job is to design, develop, and maintain data-base applications for any Tandem computer system.

Throughout this document, all references to NonStop systems indicate the software that runs on Tandem NonStop II processors or NonStop TXP processors. This manual also applies to older, NonStop 1+ processors.

A disc volume on a NonStop system can operate under either the DP2 disc process or the DP1 disc process, which is an enhanced version of the disc process used on all NonStop 1+ systems. References to DP1 can apply either to NonStop systems or to NonStop 1+ systems but references to DP2 apply to NonStop systems only.

Strictly speaking, the names DP1 and DP2 refer only to disc processes. The phrases "DP1 file" and "DP2 file" are used, however, to indicate files structured under a specific kind of disc process.

Section 1 of this manual summarizes the product. Every ENSCRIBE programmer and administrator should read this section at least once.

Section 2 describes, in detail, each of the four types of files supported by ENSCRIBE: key-sequenced files, relative files, entry-sequenced files, and unstructured files.

Section 3 summarizes the file-system and sequential I/O procedures which are available for manipulation of ENSCRIBE files.

Section 4 describes how to create ENSCRIBE disc files.

Section 5 describes how to access ENSCRIBE files.

Section 6 describes how to load data into ENSCRIBE files.

Appendix A shows the ASCII character set.

Appendix B describes the block format of ENSCRIBE-structured files, under both DP1 and DP2 disc processes.

Appendix C is a chart of major differences between the DP1 and DP2 disc processes.

SECTION 1

INTRODUCTION TO ENSCRIBE

ENSCRIBE provides high-level access to, and manipulation of, records in a data base. As an integral part of the GUARDIAN Operating System, distributed across two or more processors, ENSCRIBE helps ensure data integrity if a processor module, I/O channel, or disc drive fails.

Some of ENSCRIBE's important features are:

- Four disc-file structures: key-sequenced, relative, entry-sequenced, and unstructured
- Multiple-volume (partitioned) files
- Multi-key access to records
- Relational access among files
- Optional automatic maintenance of all keys
- Optional compression of data, index, and audit-checkpoint records
- Support of TMF's transaction auditing
- Record-locking and file-locking
- Cache buffering
- Sequential-access buffering option
- NonStop disc processes
- Mirrored discs

BASIC CONCEPTS

Understanding of these basic concepts is essential for any ENSCRIBE programmer:

- A file is a collection of related records.
- A record is a collection of one or more data items.
- A key is a value associated with a record (a record number, for example) or contained in a record (as a field) that can be used to locate one record or a subset of records in a file.
- Each record in an ENSCRIBE structured file is uniquely identified by the value of its primary key.
  - -- For key-sequenced files, the primary key is a byte field within the record and determines where a record is added to a file. The primary-key field for a key-sequenced file is defined when the file is created.
  - -- For relative files, the primary key is a record number.
  - -- For entry-sequenced files, the primary key is the relative byte address of the record.
- An alternate key is a byte field within a record that can be used to provide a logically independent access path through a file. The values of an alternate key can be used to identify a subset of records in an access path. A file's alternate-key fields are defined when the file is created. Any ENSCRIBE structured file type can have 0 to 255 alternate-key fields. Alternate key values may or may not be unique.

FILE IDENTIFIERS

You use a symbolic file identifier whenever you create, purge, rename, read, or write to a disc file. You assign this file identifier when you create the file.

File identifiers have two forms: external and internal. The external form is used when entering file identifiers from outside the file system (for example, by a user to identify a file to the GUARDIAN Command Interpreter). The internal form is used within the system when file names are passed between application processes and the operating system.

External File Identifiers

Within a system, the file identifier has three parts:

1) A volume name to identify a particular disc pack in the system

2) a subvolume name to identify the disc file as a member of a related set of files on the volume (as defined by the application)

3) a file name to identify the file within the subvolume.

This structure is illustrated in Figure 1-1.

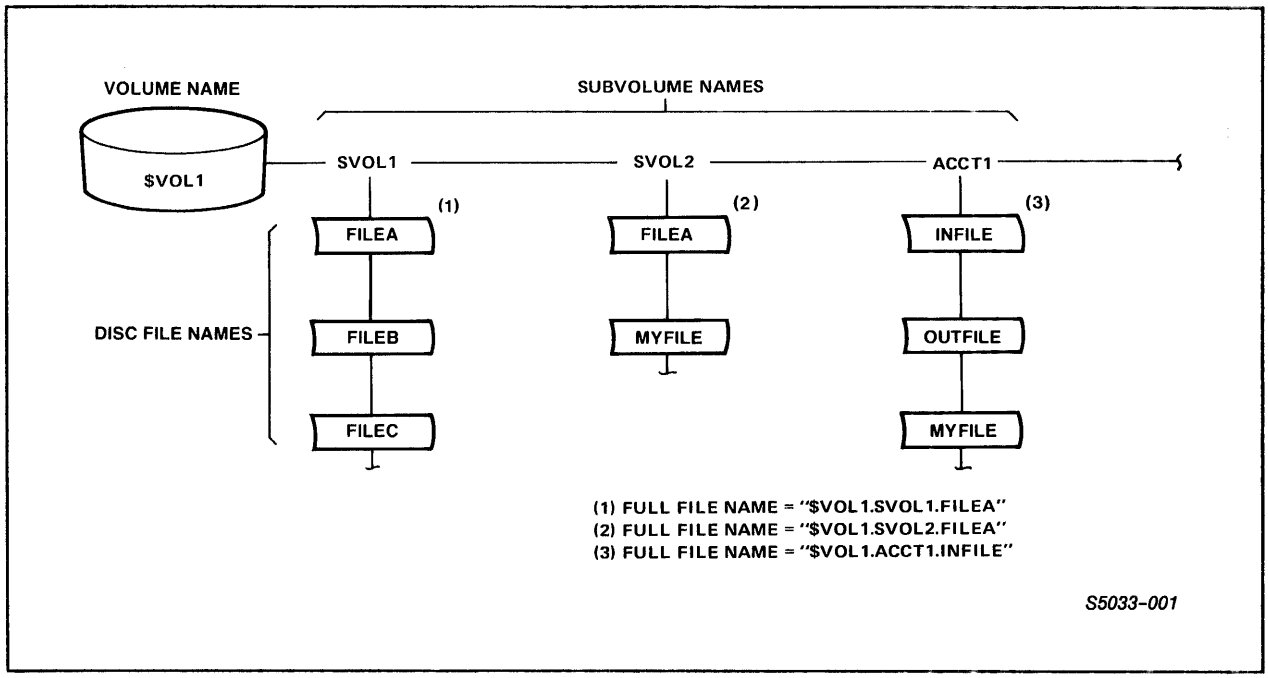


Figure 1-1. Elements of File Identifiers

In a multisystem network, the file identifier may also include a system name. For example, \PARIS.$DATA.JONES.JAN12 identifies the file named JAN12, in subvolume JONES, on volume $DATA, in system \PARIS.

For I/O purposes, the system considers processes, peripheral devices, etc., to be "files," but their identifiers do not have subvolume or file-name components. For example, a process within the same system might be named $PROC1 and one in another system (in a network) might be named \LONDON.$PROC2.

Internal File Identifiers

The internal form of a file identifier is:

```
<file name> ! 12 words, blank-filled.

where

   to access a permanent disc file, use

      file name[0:3]  =  $<volume-name><blank fill>
      file name[4:7]  =  <subvolume-name><blank fill>
      file name[8:11] =  <disc-file-name><blank fill>

   to access a temporary disc file, use

      file name[0:3]  =  $<volume-name><blank fill>
      file name[4:11] =  the <temporary-file-name> (which
                         is blank-filled) returned by CREATE

   to access a non-disc device, use

      file name[0:11] =  $<device-name> or
                         $<logical-device-number>

   to use READ or READUPDATE on another process, use

      file name[0:11] =  $RECEIVE <blank fill>

   to use WRITE or WRITEREAD on another process, use

      file name[0:11] =  $<process-identifier><blank fill>

   to write to the operator console, use

      file name[0:11] =  $0 <blank fill>
```

For network-distributed access, see "Network File Identifiers" below.

The conversion from external to internal form is performed automatically by the GUARDIAN Command Interpreter for the IN and OUT file parameters of the RUN command. (See the COMINT section of the GUARDIAN Operating System Utilities Reference Manual for details of the RUN command.) The FNAMEEXPAND procedure is provided for general conversion of file names from the external to the internal form. The FNAMECOLLAPSE procedure is provided for conversion from internal to external form.

Permanent Disc-File Identifiers

The internal form of a permanent disc-file identifier is:

```
word: [0:3]            [4:7]              [8:11]
      $<volume name>   <subvolume name>   <file name>
```

A unique volume name identifies each disc pack in the system. The name is assigned during system generation or when a new disc pack is introduced into the system. A <volume name> must be preceded by a dollar sign ($) and consists of one to seven alphanumeric characters, the first of which must be alphabetical.

A subvolume name identifies a subset of files on one disc and is assigned programmatically when a disc file is created. A <subvolume name> consists of one to eight alphanumeric characters; the first character must be alphabetical.

A disc file name identifies a specific disc file and is assigned programmatically when the disc file is created. A <disc file name> consists of one to eight alphanumeric characters; the first character must be alphabetical.

For example,

```
INT .file^name[0:11] := "$STORE1 ACCT1   MYFILE  ";
```

is a valid identifier for a permanent disc file.

Temporary File Identifiers

The CREATE procedure assigns a temporary file identifier when it creates a temporary file. A <temporary file name> consists of a crosshatch (#) followed by four numerical characters. To specify a temporary file on volume $STORE1, for example, you might use

```
INT .file^name[0:11] := ["$STORE1 ", 8 * ["  "]];
CALL CREATE(file^name);
```

Only the <volume name> is supplied. CREATE returns the <temporary file name>.

Network File Identifiers

A file identifier can include a <system number> that identifies a file as belonging to a particular system on a network. (See the EXPAND Reference Manual for information regarding networks of Tandem systems.)

In this context, a file identifier beginning with a dollar sign ($) is said to be in local form, to distinguish it from a file identifier beginning with a backslash (\) which characterizes the network form. Specifically, the network form is:

```
<network file identifier>  ! 12 words, blank-filled

   word[0].<0:7>  = \ (ASCII backslash)
   word[0].<8:15> = <system number>, in octal
   word[1:3]      = <volume name>, <device name>, or
                    <process identifier>
   word[4:11]     = same as in local file identifier

where

   <system number>

      is an integer between 0 and 254 that designates a
      particular system.  System numbers are assigned during
      system generation (SYSGEN).

   <volume name>

      consists of one to six alphanumeric characters, the
      first of which must be alphabetic.
```

Note that the name of a disc volume or other device, when embedded within a network file identifier, can have no more than six characters and does not begin with a dollar sign. Similar restrictions apply to the network form of process identifier; the <process identifier> in words 1 and 2 can have no more than four characters (the first one must be alphabetic, as usual) and does not include the initial dollar sign.

The application program rarely, if ever, concerns itself with octal system numbers in network file identifiers. Usually, the application passes the external form of the file identifier (which contains a system name, rather than a number) to the procedure FNAMEEXPAND, which converts the system name into the corresponding number.

Network file identifiers are converted between internal and external forms by the procedures FNAMEEXPAND and FNAMECOLLAPSE.

DISC-FILE ORGANIZATION

A disc file must be created before it can be accessed. You can create a file by calling the CREATE procedure or by using the File Utility Program (FUP)'s CREATE command. When created, a file can be designated as either permanent or temporary. A permanent file remains in the system after access is terminated; a temporary file is deleted when access is terminated.

You also specify the file's type when you create it. ENSCRIBE supports four file types: key-sequenced, relative, entry-sequenced, and unstructured files. Taken as a group, key-sequenced, relative, and entry-sequenced files are known as structured files. The facilities available with structured files differ significantly from those available with unstructured files. Each of the four file types is described briefly below, beginning with the structured files. Section 2, "File Structures," discusses them in more detail.

Physical storage for a disc file is allocated by the file system in the form of discontiguous file extents. A file extent itself is a contiguous block of storage, starting on a sector boundary and containing a multiple of 2,048 bytes--one page.

With the DP1 disc process, a file can have as many as 16 extents. A DP2 partitioned file is also limited to 16 extents per partition. The maximum number of extents in a nonpartitioned DP2 file is limited only by the maximum label size and the number of alternate keys (see Appendix C), which normally allow more than 900 extents to a file. Within this limit, you can use the MAXEXTENTS attribute to set an arbitrary limit for any DP2 file.

In any case, the first extent is designated the primary extent and can differ in size from the remaining secondary extents. This allows a file to be created with a large primary extent, to contain all the data to be initially placed in the file, and smaller secondary extents to use minimal increments of disc space as the file grows.

An application process can allocate one or more extents in an open file via a call to CONTROL, with <operation> = 21. CONTROL can also deallocate unused extents.

Some further information about extent sizes is in the "Block Sizes and Extents" discussion in Section 4.

CHARACTERISTICS COMMON TO ALL ENSCRIBE FILES

For all ENSCRIBE files:

- An application process can remove all data from a file, without deleting the file, by use of the CONTROL procedure's "purge data" operation.
- File-locking procedures are available to coordinate access to a sharable file.
- The disc process automatically retries parity and overrun errors.
- The file system automatically retries communication-path errors if the file is open with a "synchronization depth" greater than zero. Return of a path error in this case indicates that the file is no longer accessible.
- The maximum number of files in a volume is a function of system configuration.

STRUCTURED FILES

All data transfers between an application process and a structured disc file are done in terms of logical records. Placement of (and access to) records in a disc file is determined by the file structure, which is specified when the file is created.

For structured files, the maximum length of a logical record (that is, the maximum number of bytes that can be inserted in a single operation) is specified for each file when the file is created. The actual number of bytes in a logical record can vary up to the specified record length; the minimum number of bytes depends on the file structure.

Each record has a length attribute. The length attribute is a count of the number of bytes inserted when the record was written. A record's length is returned when the record is read.

Several utilities, such as DDL, ENFORM, and ENABLE, exist to aid in use of structured files.

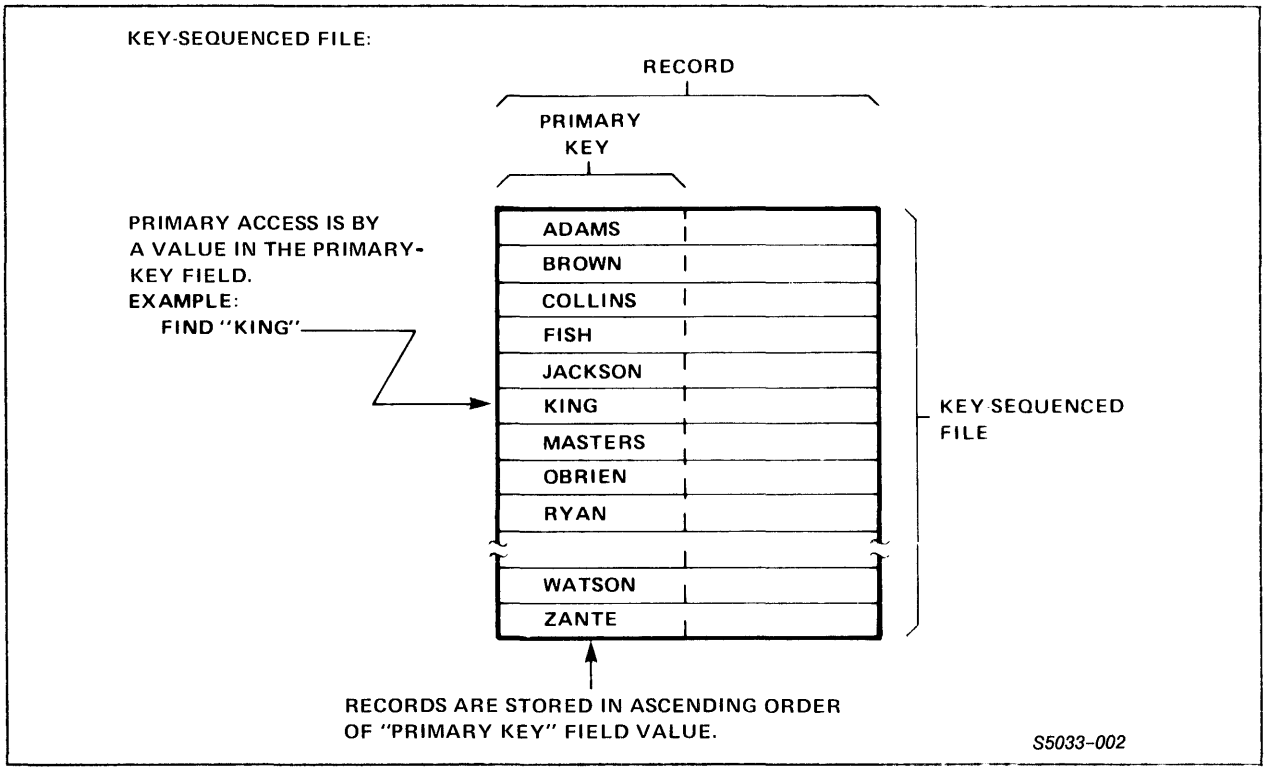


Figure 1-2. Key-Sequenced File Structure

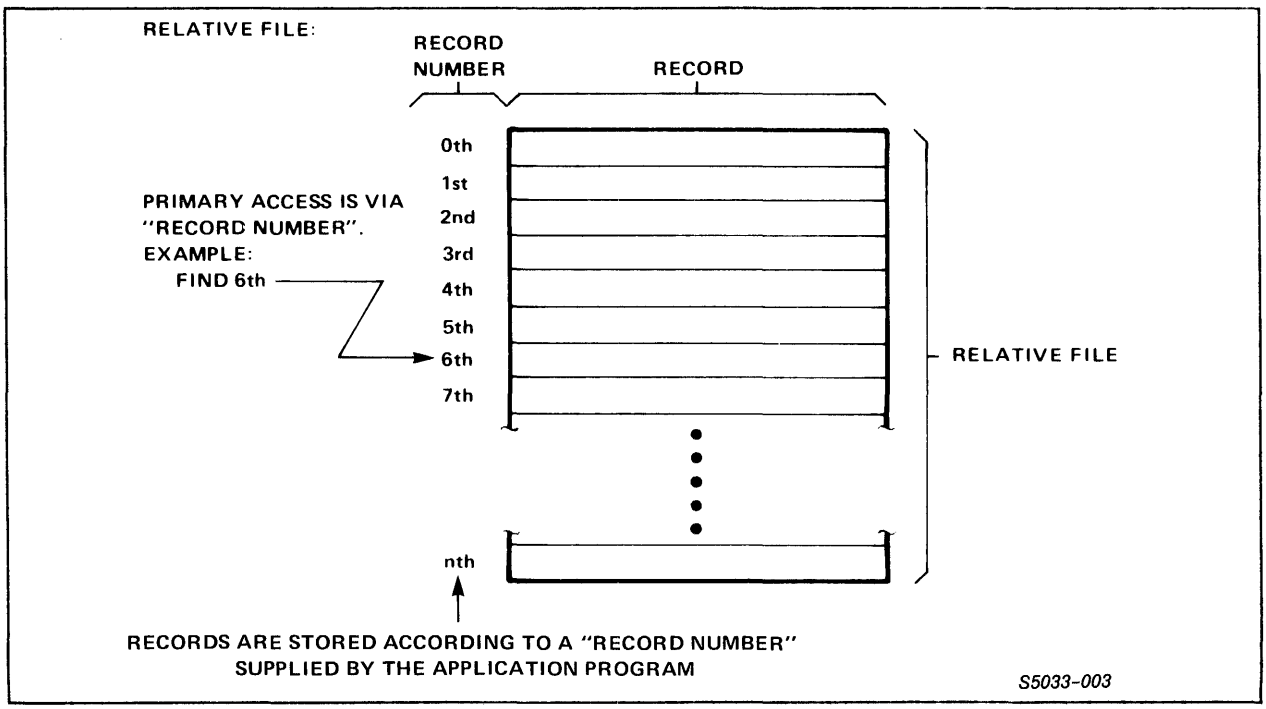


Figure 1-3. Relative File Structure

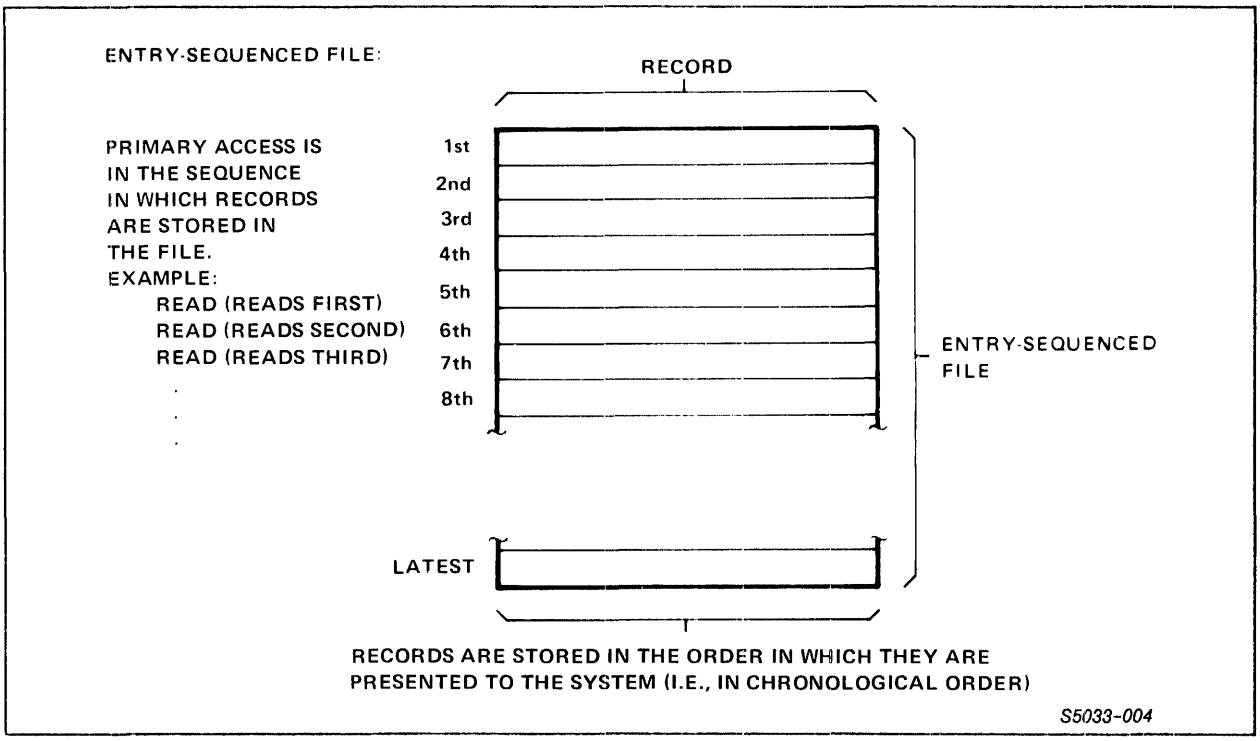


Figure 1-4. Entry-Sequenced File Structure

Key-Sequenced File Structure

Figure 1-2 illustrates the structure of a key-sequenced file. Records are stored in ascending sequence, ordered by the value of a field within each record called the primary key field. The primary key field is designated when a key-sequenced file is created. It can be any set of contiguous bytes within the data record. Physical and logical record lengths can be variable; a record occupies only the amount of space specified for it when inserted into the file.

Relative File Structure

Figure 1-3 illustrates the structure of a relative file. Records are stored in a position relative to the beginning of the file, according to a record number supplied by the application program. A record number is an ordinal value and corresponds directly to a physical record position in a file. Each physical record position in a relative file occupies a fixed amount of space (although logical record lengths may be variable).

Entry-Sequenced File Structure

Figure 1-4 illustrates the structure of an entry-sequenced file. Records are appended to the end of an entry-sequenced file in the order in which they are presented to the system. Once added to a file, a record's contents can be updated but the record's size cannot be changed and the record cannot be deleted (although an application program may use a field within the record to indicate that it has been logically deleted). Physical and logical record lengths can be variable; a record occupies only the amount of space specified for it when inserted into the file.

Multi-Key Access to Structured Files

A record consists of one or more fields, as shown in Figure 1-5.

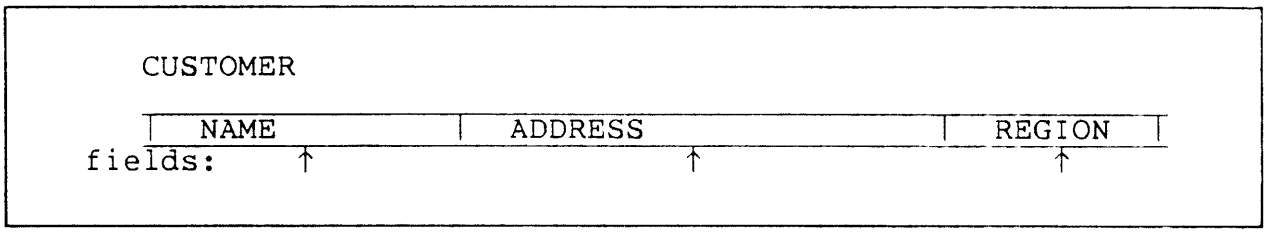


Figure 1-5. A Record, with Three Fields, in a Key-Sequenced File

Each record in a file is uniquely identified among other records in that file by the value of its primary key. For key-sequenced files, the primary key is a byte field within a record; for relative files, the primary key is a record number; for entry-sequenced files, the primary key is a record address. Records in a file are physically ordered by ascending value of the primary key, as shown in Figure 1-6.

Also, one or more byte fields within a record can be designated as alternate keys (see Figure 1-7). Any structured file can have as many as 255 alternate-key fields. Alternate-key values need not be unique.

ENSCRIBE lets you use the primary-key value to locate one unique record among other records in the same file. For example, in Figure 1-8, the primary key is the name field and the primary-key value J.A. Jones locates the only record having that name.

By using alternate-key values, ENSCRIBE lets you locate several associated records of the same type. For example, using Figure 1-8 again, the REGION field is an alternate key and the value CENTRAL locates two records.

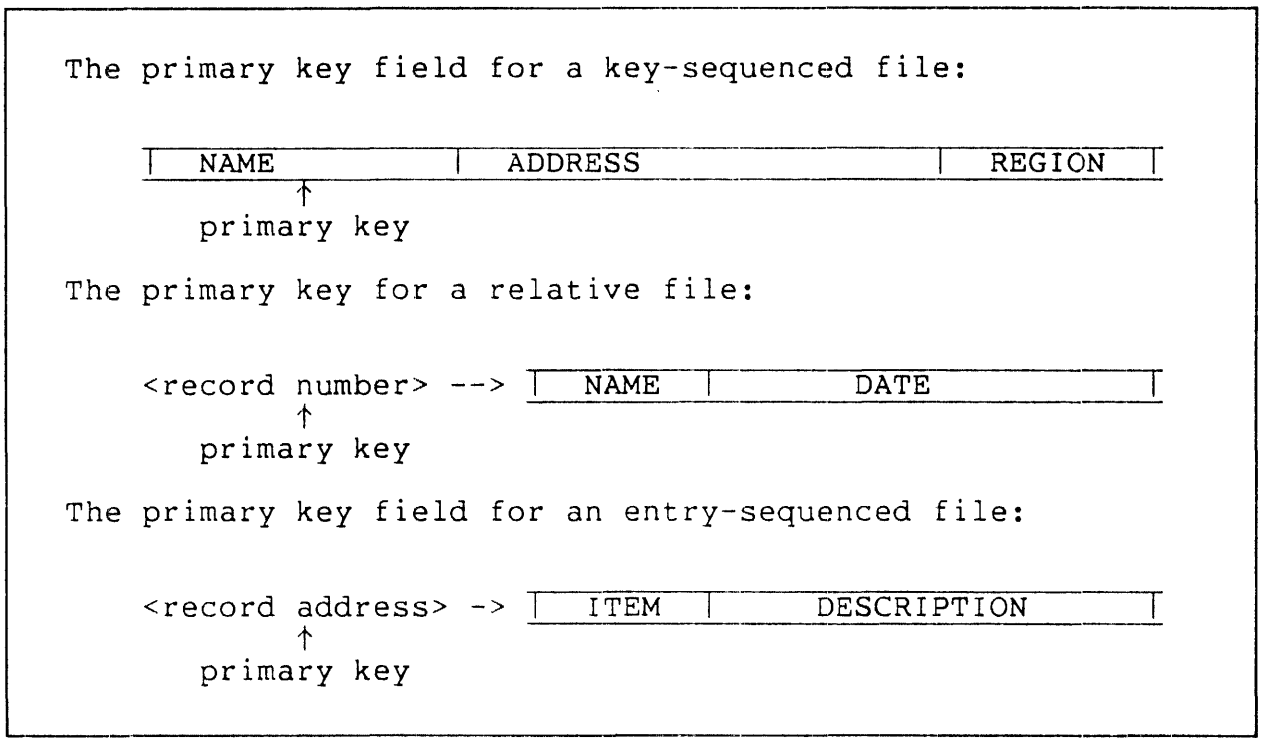


Figure 1-6. Primary Keys in Structured Files

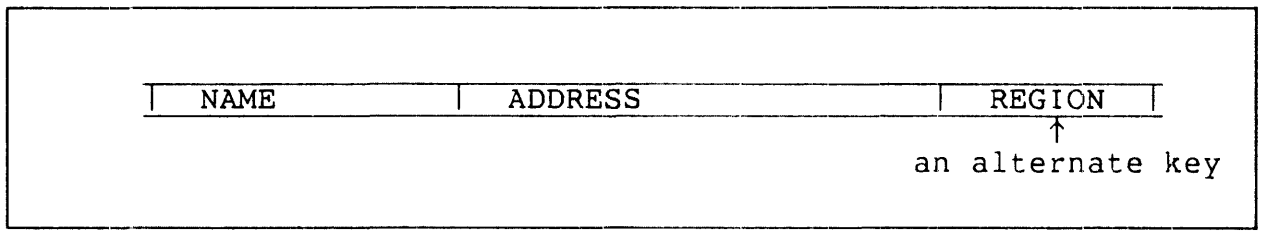


Figure 1-7. An Alternate-Key Field

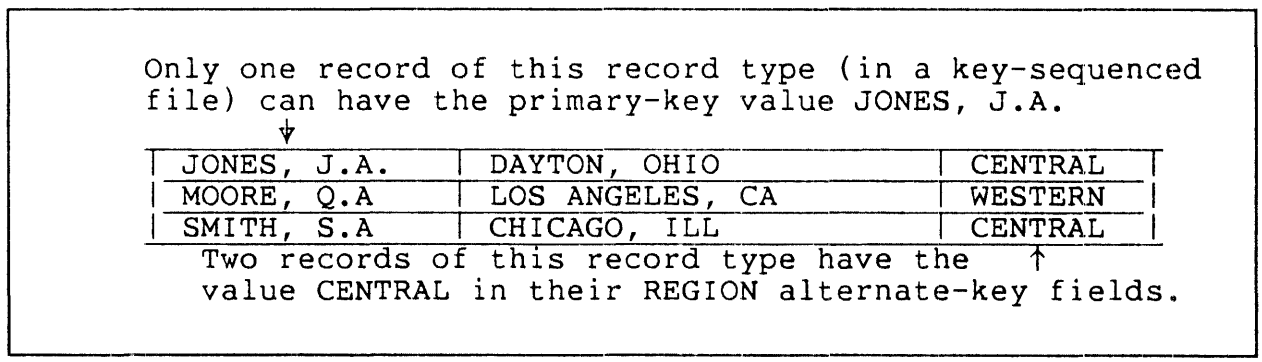


Figure 1-8. Using Key Values to Locate Specific Records

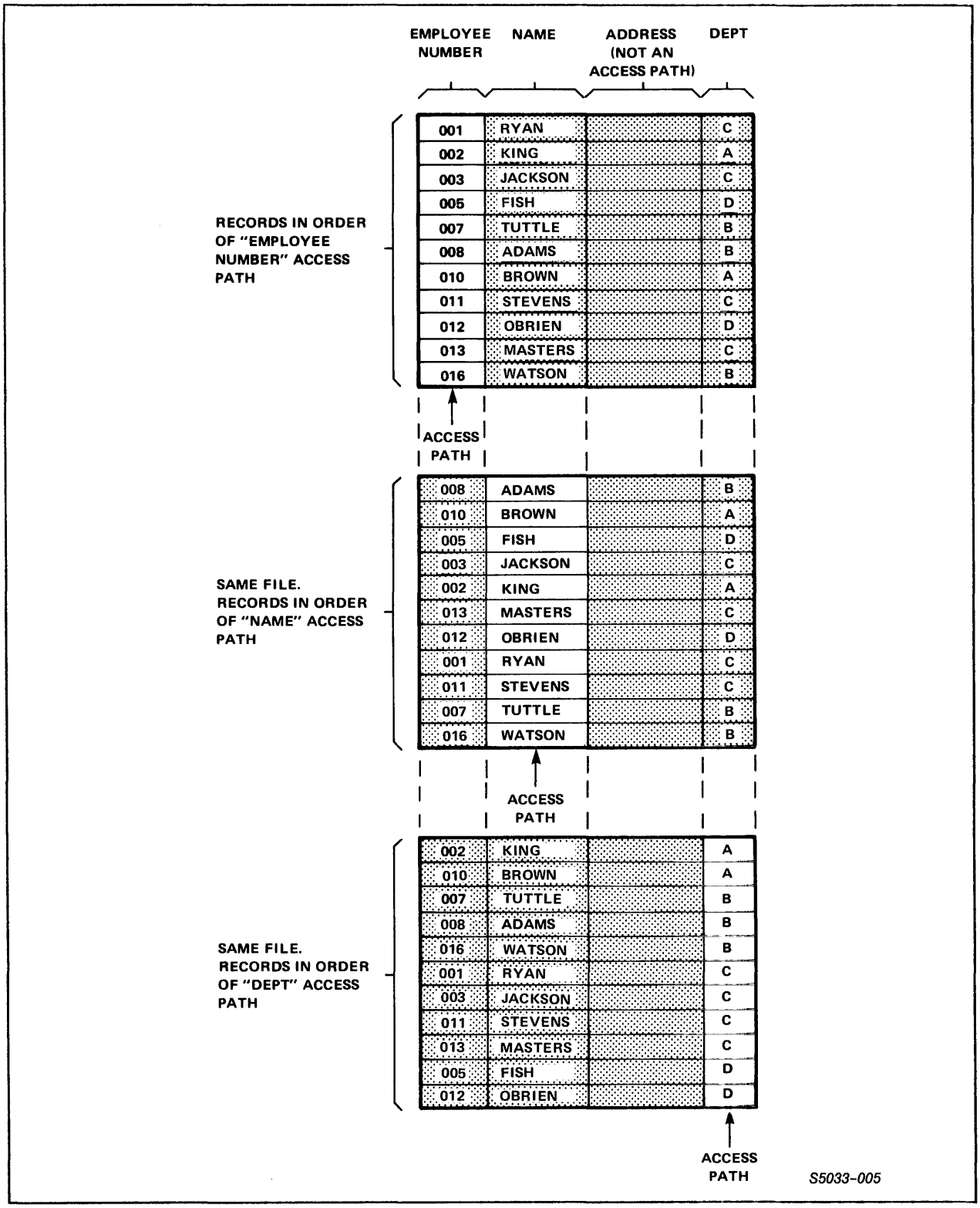


Figure 1-9. Access Paths

Each key in a structured file provides a separate access path through records in that file. Records in an access path are logically ordered by ascending access-path key values. A simple employee file with three separate access paths, provided by three different key fields, is shown in Figure 1-9.

A subset of records in a designated access path can be described by a positioning mode and a key value. The positioning modes are approximate, generic, and exact. The approximate mode selects all records whose access-path key values are equal to or greater than the supplied key value. The generic mode selects all records whose access-path key value matches a supplied partial value. The exact mode selects only those records whose access-path key value matches the supplied key value exactly. Examples of subsets returned with these three positioning modes are shown in Figure 1-10.

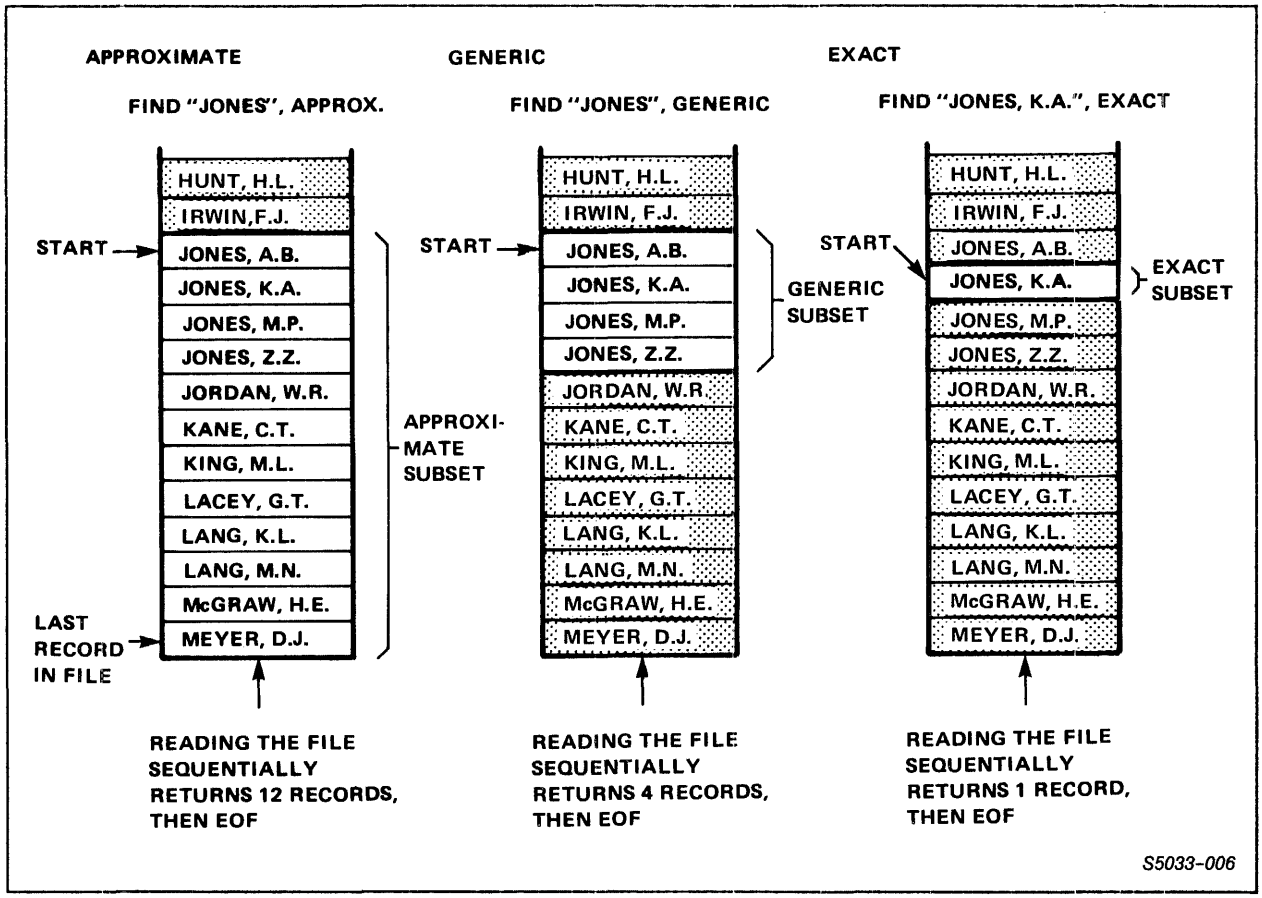


Figure 1-10. Approximate, Generic, and Exact Subsets

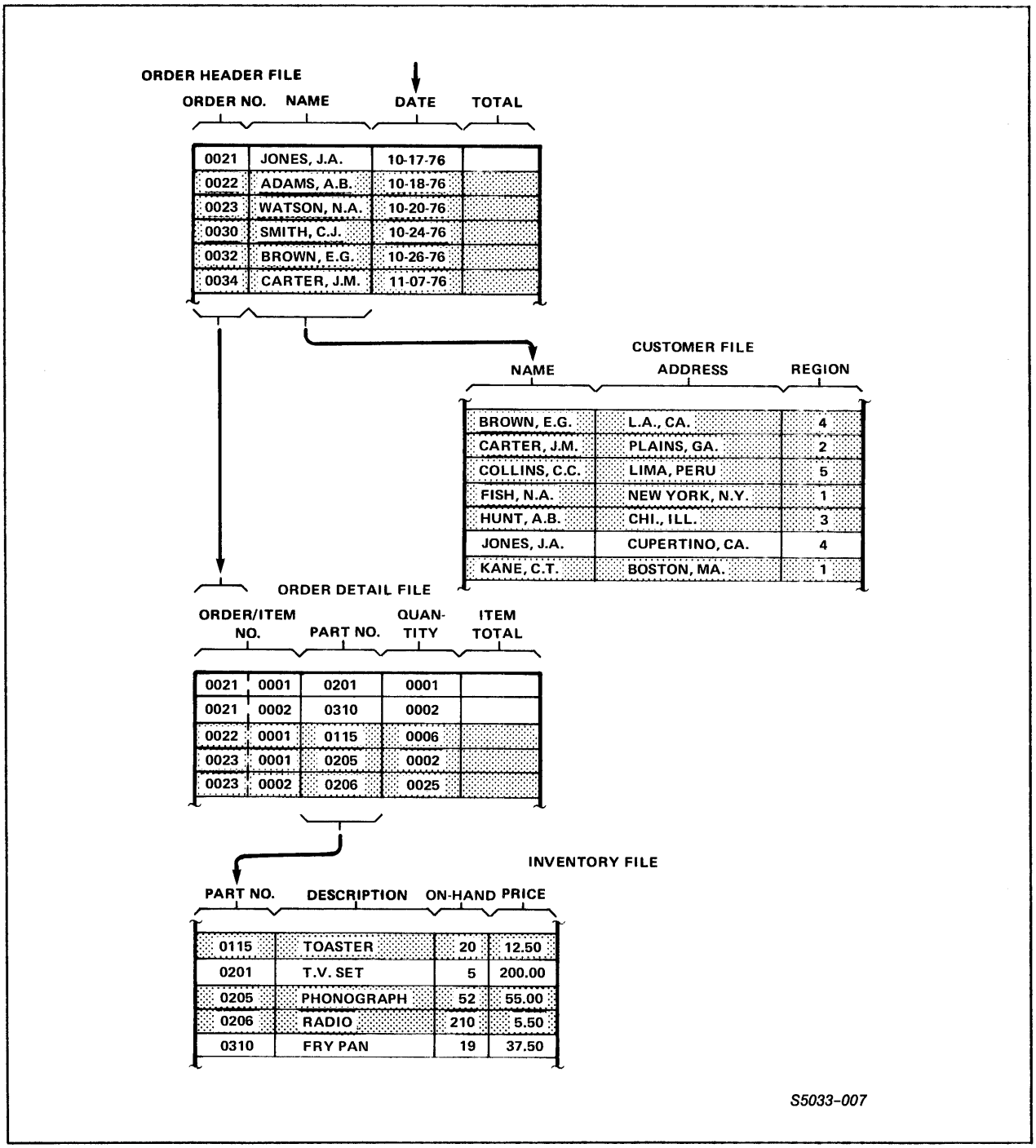


Figure 1-11. Relational Access among Structured Files

Relational Access

Relational access among structured files in a data base is accomplished by obtaining a value from a field in a record in one file and using that value to locate a record in another file. An example of relational access is shown in Figure 1-11. ENFORM automates this access. (See the Introduction to ENFORM manual.)

Automatic Maintenance of All Keys

When a new record is added to a file or a value in an alternate-key field is changed, ENSCRIBE automatically updates the indices to the record (the value of a record's primary key cannot be changed). This operation is entirely transparent to the application program.

If more key fields are later added to a file, but existing fields in that file are not relocated, existing programs that access the file need not be rewritten or recompiled.

Data and Index Compression

For key-sequenced files, an optional data-compression technique permits storing more data in a given disc area, thereby reducing the number of head repositionings.

Similarly, an optional index-compression technique is provided for key indices to data records.

Data and index compression can be specified for a file when the file is created. (See "Compression" in Section 4).

With the DP2 disc process, audit-checkpoint records can be compressed to increase I/O efficiency and to decrease TMF audit-trail consumption. You can set this feature when you create the file or you can alter it later by using the SETMODE procedure or the File Utility Program (FUP) command ALTER. (See the "Audit-Checkpoint Compression" discussion in Section 5.)

UNSTRUCTURED FILES

An unstructured disc file is essentially a byte array. It is normally used as a code file or EDIT file, not as a data file. The organization of an unstructured disc file (the lengths and locations of records within the file) is the responsibility of the application process.

Data stored in an unstructured file is addressed in terms of a relative byte address (RBA). A relative byte address is an offset, in bytes, from the first byte in the file; the first byte is at RBA zero.

Associated with each open file are three pointers: a current-record pointer, a next-record pointer, and an end-of-file pointer.

When a file is opened, the current-record and next-record pointers are set to point to the first byte in the file. A read or write operation always begins at the byte pointed to by the next-record pointer. The next-record pointer is advanced with each read or write operation by the number of bytes transferred; this provides automatic sequential access to a file. Following a read or write operation, the current-record pointer is set to point to the first byte affected by the operation.

The next-record and current-record pointers can be set to an explicit byte address in a file, thereby providing random access. The end-of-file pointer contains one plus the relative byte address of the last byte in a file. The end-of-file pointer is automatically advanced by the number of bytes written when a record is appended to the end of a file.

MULTIPLE-VOLUME (PARTITIONED) FILES

When a file is created, the file can be designated to reside entirely on a single volume or it can be partitioned to reside on separate volumes. Moreover, the separate volumes need not reside on the same system; a file can be partitioned accross network nodes. As many as 16 partitions are permitted, each with as many as 16 extents.

In addition to providing a much larger maximum file size, the use of partitioned files provides for simultaneous access to a file's records:

- If the file resides on several volumes connected to the same control device, seeking (disc-head repositioning) can be occurring on all volumes simultaneously.
- If each file resides on a volume that is connected to a different control device, several data transfers (as well as seeks) with the file can occur concurrently.
- If each volume's control device is connected to a different processor module, simultaneous processing of the file's data can occur, as well as simultaneous seeking and data transfers.

FILE DIRECTORY

A disc volume's file directory holds information about all the files on that volume. You govern the size of this directory, either during system generation or when using the Peripheral Utilities Program (PUP) to label the disc, by estimating how many files you want it to hold. The system translates this to an approximate extent size when it creates the actual directory file. The actual number of files that will fit in a directory extent varies according to the types of files involved, because some file types need larger file labels than other types. Therefore, the actual capacity may not be precisely what you specified.

With the DP1 disc process, the directory has only one extent. If you create enough files to overflow that extent, an error is returned. Therefore, when you estimate the size of a DP1 directory, you should provide space for the greatest possible number of files.

The DP2 disc process, however, can create as many as 987 directory extents, so the creation of too many files merely causes the disc process to allocate another directory extent.

AUDITED FILES

In a system with TMF, any data-base file can be designated to be an audited file. To help maintain data-base consistency, the Transaction Monitoring Facility (TMF) audits all transactions involving files designated as audited files. That is, TMF maintains images (in an audit trail) of the data-base changes wrought by those transactions. If necessary, it can use the audit trail later to back out failed transactions or to restore audited files that some system failure has rendered inconsistent. With the DP2 disc process, audited files and audit trails cannot coexist on the same volumes.

TMF also uses a record-locking mechanism to perform concurrency control for audited files. This ensures that none of a given transaction's changes are visible to other, concurrent transactions until all the given transaction's changes are committed.

COORDINATION AMONG MULTIPLE ACCESSORS

Several different processes can have access to one file at the same time. For coordination of simultaneous access, each process must indicate (when opening the file) how it intends to use the

file. Both an access mode and an exclusion mode must be specified.

The access mode specifies the operations to be performed by an accessor. The access mode is specified as either read/write (default access mode), read-only, or write-only.

The exclusion mode specifies how much access other processes will be allowed. It can provide shared, exclusive, or protected access, as described in the GUARDIAN Operating System Programmer's Manual description of the OPEN procedure.

LOCKING

The access and exclusion mode operate on a file from the time it is opened until the time it is closed. To prevent concurrent access to a disc file for shorter periods of time, two locking mechanisms are provided: file-locking and record-locking. TMF enforces a special set of locking rules for audited files.

Locking is discussed in more detail in Section 5, "File Access".

"WAIT" I/O AND "NO-WAIT" I/O

ENSCRIBE can let an application process execute concurrently with its file operations, via "no-wait" I/O.

The default is "wait" I/O; when designated file operations are performed (via file-system procedure calls), the application process is suspended, waiting for the operation to complete.

"No-wait" I/O means that, when designated file operations are performed, the application process is not suspended. Rather, the application process executes concurrently with the file operation. The application process waits for an I/O completion in a separate file-system procedure call.

"Wait" and "no-wait" I/O are described in the GUARDIAN Operating System Programmer's Guide. See also the descriptions of the OPEN, READ, and AWAITIO procedures in the System Procedure Calls Reference Manual.

BUFFERING

Cache buffering and sequential block-buffering can help make I/O operations more efficient.

Cache

The cache, or buffer pool, is an area of main memory reserved for buffering blocks read from disc. With the DP2 disc process, you use the PUP commands SETCACHE and LABEL to specify the cache size. With DP1, the cache size is specified during system generation.

When a process reads a record, ENSCRIBE first checks the cache for the block that contains the record. If that block is already in the cache, the record is transferred from the cache to the application process. If the cache does not contain the block, the block is read from the disc into the cache, then the requested record is transferred to the application process.

If no space is available in the cache when a block must be read in, a least-recently-used (LRU) algorithm determines which block to overlay. The purpose of this algorithm is to keep the most recently used blocks in main memory whenever possible. (For DP1 key-sequenced files, the algorithm is weighted to favor index blocks.)

When a process writes a record, what happens differs according to the type of disc process and the options used in opening the file. Under the DP2 disc process if the buffered cache feature is not used, or under DP1 for a nonaudited file, the cache block that contains the record is modified then immediately written to disc. (If the block to be modified is not in the cache, it is first read from the disc.) However, the modified block remains in the cache until the buffer space is needed for overlay. This is called write-through cache.

Under DP2, you can open a file with buffered cache, so the cache contents are written to disc, or flushed, less frequently. If several data changes occur to records in the same block in the cache, transaction time is faster because less I/O to the disc is required. On the other hand, data-base changes do not get into the actual disc file until the cache block is flushed for some reason. These situations cause a block to be flushed:

- Any opener closes the file.
- The SETMODE procedure forces flushing.
- The control-point mechanism of TMF forces flushing.

- Space is needed for a new block to be read into cache. The disc process selects the least recently used block and flushes it to make room for the new one.

- Some aspect of cache configuration, such as an unstructured file's buffer size, is changed dynamically.

- The REFRESH procedure, or equivalent PUP command, is used on the file.

- When the disc process has been idle for a sufficient period of time, it uses the free time to flush modified cache buffers until it receives a user request.

Audited files always use buffered cache. DP2 nonaudited files can use either buffered or write-through cache.

Write-through cache is the default for nonaudited files, because a system failure or disc process takeover (with <sync depth> = 0) could cause the loss of buffered updates and an unmodified application may not detect or handle such a loss properly. Such a loss of buffered updates would be indicated by error 2. Buffered cache is used for audited files because TMF can recover committed, buffered updates lost due to a system failure.

DP2 avoids fragmentation of cache memory space, even if files of all different block sizes are buffered, by grouping all 4096-byte blocks in one area, all 2048-byte blocks in another area, etc. You set the amount of cache memory devoted to each block size during system configuration. You can alter this arrangement dynamically by using the PUP command LABEL.

Sequential Block-Buffering (Structured Files Only)

If a program reads a file sequentially, its access time to individual records can be greatly reduced if it opens the file with sequential block-buffering. Basically, this option allows the record-deblocking buffer to be in the application process's data area (rather than in a disc process). ENSCRIBE then uses this buffer to deblock the file's records. The advantage of this buffering is that it eliminates the request to the disc process to retrieve each record in a block. Instead, a request retrieves an entire block of records.

If sequential block-buffering is to be used, the file usually should be opened with protected or exclusive access. Shared access can be used, although it can cause some problems.

Sequential block-buffering is discussed in detail in Section 5, "File Access."

OPERATIONS ON FILES

Common file operations include creating files, describing record formats, loading files, refreshing file control block (FCB) information, purging data, and viewing data.

Creating Files

Disc files are created (defined) by either calling the file system's CREATE procedure or using the File Utility Program (FUP):

- Programmatic creation of disc files is accomplished by supplying the appropriate parameters to the CREATE procedure.
- The FUP commands SET, RESET, and SHOW let you specify, display, and modify creation parameters (file type, record length, key description, etc.) before actually creating a file. If you like, you can set the creation parameters to be like another, existing file. FUP's CREATE command then creates a file with the currently set parameters. The ALTER command allows changing some of those parameters after the file is created. FUP accepts commands from an online terminal or from a command (OBEY) file.

File creation is described in Section 5, "File Creation."

Describing Record Formats (DDL)

The Data Definition Language (DDL) provides a uniform method of describing record formats, regardless of the programming language used (COBOL, FORTRAN, or TAL) to access the record. DDL also provides a system-wide definition of record formats so all programs have a consistent definition of a given record format. (See the Data Definition Language Reference Manual.) In addition to programming-language source code, DDL can produce FUP file-creation commands for data-base files which are then accessible through ENSCRIBE. File creation is described in Section 5, "File Creation."

Loading Files

You can use the File Utility Program (FUP) to load data into existing ENSCRIBE files. You specify the set of records to be loaded and the file's data- and index-block loading factor. (The loading factor determines how much free space to leave within a block). FUP attempts to optimize access to a file by placing the lowest-level index blocks on the same physical cylinder as their associated data blocks, thus reducing the amount of head repositioning.

File-loading is described in Section 6, "File Loading".

Refreshing FCB Information

The information in an open file's file control block (FCB), such as the end-of-file (EOF) pointer, is kept in main memory. To maximize performance, the EOF pointer normally is not written to the file's disc label each time the EOF is advanced.

At the expense of some processing throughput, however, you can specify when creating a file that the disc label should be refreshed automatically each time the FCB information changes. This autorefreshing is always on for DP2 key-sequenced files; any REFRESH setting is ignored.

Alternatively, you can use the REFRESH procedure or an equivalent Peripheral Utility Program (PUP) command to update the disc label. (See "End-Of-File Pointer" and "Refreshing" in Section 5.)

Purging Data

Either the File Utility Program (FUP) command PURGEDATA or the CONTROL procedure's "purge data" operation can logically--but not physically--remove all data from a file by resetting pointers to relative byte 0. Also, either the PURGE procedure or the FUP command PURGE can delete a file from the disc directory.

If you have set the CLEARONPURGE flag for a file, using either function 1 of the SETMODE procedure or the FUP command SECURE, all data will be physically erased (overwritten with zeros) when the file is deleted (see "Purging Data" in Section 5). CLEARONPURGE has no effect after execution of PURGEDATA, however.

Viewing Data

ENFORM is the query language and report writer designed for ENSCRIBE data files. See the Introduction to ENFORM manual.

Table 1-1. Record-Management Functions Summary

Function	Description	Procedure
Find	Set the current position, access path, and positioning mode for a file. This can indicate the start of a subset of records in anticipation of reading the set sequentially, or it can specify a record for subsequent updating.	KEYPOSITION, POSITION
Insert	Insert a new record into a file according to its primary-key value.	WRITE
Read	Read a subset of records sequentially.	READ
Update	Update a record in a random position in a file.	READUPDATE, WRITEUPDATE
Delete	Delete the record in a key-sequenced or relative file as indicated by a primary-key value.	WRITEUPDATE
Lock	Lock the whole file or just the current record.	LOCKREC, LOCKFILE, READLOCK, READUPDATELOCK
Unlock	Unlock the current record in a file or all records in the file.	UNLOCKREC, UNLOCKFILE, WRITEUPDATEUNLOCK
Define	Define a new file.	CREATE

GENERATING APPLICATIONS

ENABLE is a software product that automatically generates applications for ineractively updating and retrieving records in an ENSCRIBE data file. ENABLE-generated applications can be used

for temporary requirements or as modules in custom applications. The ENABLE User's Guide describes how to use this product.

RECORD-MANAGEMENT FUNCTIONS

Manipulation of records in an ENSCRIBE file is performed by calling file-system procedures. Record-management functions and their associated procedures are summarized in Table 1-1. The procedures listed there are described in the System Procedure Calls Reference Manual.

FILE-SYSTEM IMPLEMENTATION

Internal operation of the file system is described thoroughly in the GUARDIAN Operating System Programmer's Guide. The ENSCRIBE programmer may find it useful to be familiar with that information. In particular, the programmer should understand the action that the file system takes when a communication-path failure occurs and the corresponding action that the application program must take to recover.

SECTION 2

FILE STRUCTURES

ENSCRIBE provides these four disc-file structures:

- Structured files: key-sequenced, relative, and entry-sequenced
- Unstructured files

This section lists some characteristics unique to unstructured files, then describes the three structured-file types and their basic access concepts. Finally, it describes alternate keys, which apply to all three types of structured files.

UNSTRUCTURED FILES

This subsection lists some characteristics of unstructured disc files.

EDIT Files

Files created by the EDIT utility are unstructured files. Such structure as they have is imposed by EDIT, not ENSCRIBE. These files can be read by the EDITREAD procedure, by the sequential I/O (SIO) routines, or by EDIT. EDIT files are identifiable by their file-type code, 101.

File Pointers and Relative Byte-Addressing

Data access in an unstructured disc file is via a relative byte address. Three file pointers are associated with each open unstructured disc file: a next-record pointer, a current-record

pointer, and an end-of-file pointer. (See "File Pointers and Relative Byte-Addressing" in Section 5).

Buffer Size (DP2 Only)

When a user performs I/O on a DP2 unstructured file, the file is transparently blocked using one of the four valid DP2 block sizes (512, 1024, 2048, or 4096 bytes). The default size is 4096 bytes. You can set this transparent block size, called BUFFERSIZE, when you create the file or you can alter it later by using the SETMODE procedure or the File Utility Program (FUP) command ALTER.

DP2 performance with unstructured files is best when transfers begin on BUFFERSIZE boundaries and are integral multiples of BUFFERSIZE.

Other Characteristics

With unstructured files:

- It is the application's responsibility to determine optimum record sizes and to block records in an efficient manner.
- Data are physically located on discs in 512-byte sectors.
- For a DP1 data transfer to be possible,

  $$\text{mod}(\text{current-record-pointer}, 512) \leq 4096 - (\text{transfer-length})$$

  Thus, an attempt to transfer 3585 to 4095 bytes (inclusive) may fail if the current-record pointer is positioned such that the transfer spans more than eight sectors or three extents.

STRUCTURED FILES

Structured files are key-sequenced, relative, or entry-sequenced.

Key-Sequenced Files

A key-sequenced file consists of a set of variable-length records. Each record is uniquely identified among other records in a key-sequenced file by the value of its primary-key field. Records in a key-sequenced file are logically stored in order of

ascending primary-key values. The primary-key value must be unique and cannot be changed when updating a record.

A record can vary in length from one byte to the maximum record size specified when the file was created. With the DP2 disc process, the maximum record size for a key-sequenced file is 4062 bytes (assuming a 4096-byte block size). With DP1, at least two records must fit in a key-sequenced block, so the maximum record size is 2035 bytes. The number of bytes allocated for a record is the same as that written when the record was inserted into the file. Each record has a length attribute that is optionally returned when a record is read. A record's length can be changed after the record has been entered (with the restriction that the length cannot exceed the specified maximum record size). Records in a key-sequenced file can be deleted.

A key-sequenced file is physically organized as one or more bit-map blocks (DP2 files only) and a tree structure of index blocks and data blocks. The bit-map blocks organize the free space of a DP2 structured file (see Appendix B). With DP1, a linked list performs this function.

Each DP2 data block contains a header plus one or more data records, depending on the record size and data-block size; a DP1 data block must have room for a header plus two or more records. For each data block there is an entry in an index block containing the value of the key field for the first record in the data block and the address of that data block.

The position of a new record inserted into a key-sequenced file is determined by the value of its primary-key field. If the block where a new record is to be inserted into a file is full, a block split occurs. This means that the disc process allocates a new data block, moves part of the data from the old block into the new block, and gives the index block a pointer to the new data block.

When an index block fills up, it is split as described above: a new index block is allocated and some of the pointers are moved from the old index block to the new one. The first time this occurs in a file, the disc process must generate a new level of indices. It does this by allocating a higher-level index block containing the low key and pointer to the two lower-level index blocks (which in turn point to many data blocks). The disc process must do this again each time the "root" (highest-level) block is split.

The DP2 disc process sometimes performs a three-way block split, creating two new blocks and distributing the original block's data or pointers (plus the new record or pointer) among all three.

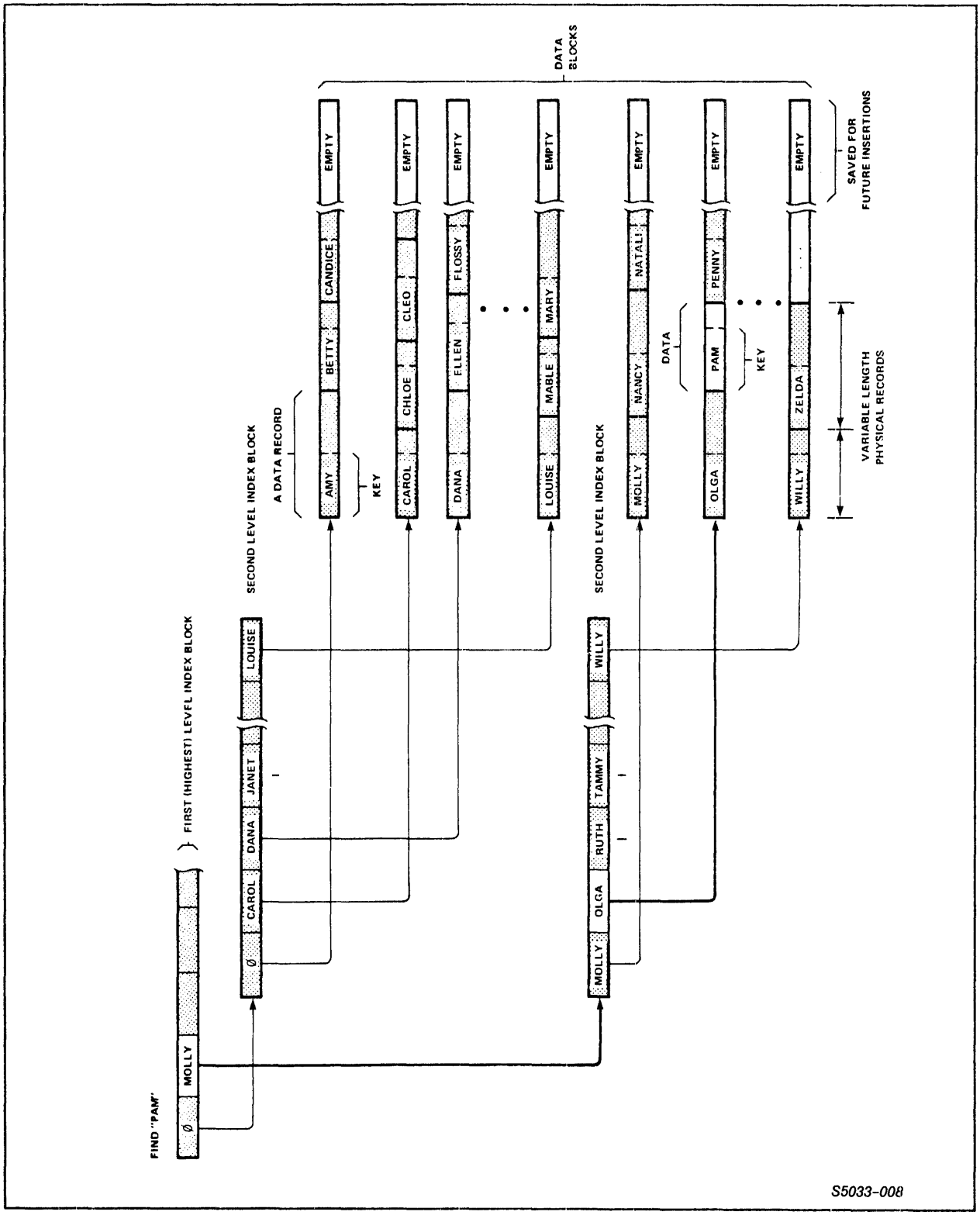


Figure 2-1. Key-Sequenced File Structure

If your record size is large, you should also use a large block size. If the block size is too small to hold more than a few records, block splits occur more frequently and disc-space usage is less efficient.

Typically, in a changing data base, most blocks will be approximately two-thirds full at any given time. When you load data into a key-sequenced file, you can specify how much slack (empty space for future growth) to provide (see the LOAD command in the FUP section of the GUARDIAN Operating System Utilities Reference Manual.

Note that data records are never chained together in ENSCRIBE key-sequenced files. Instead, the tree structure is dynamically rebalanced to ensure that any record in the file can be accessed with the same number of READ operations, that number being the number of levels of indices plus one for the data block.

An example of an application for a key-sequenced file is an inventory file where each record describes a part. The key field for that file would be the part number, so the file would be ordered by part number. Other fields in the record could contain the vendor name, quantity on hand, etc. Note that ENSCRIBE is concerned only with key fields. The content of all fields and the location within the record of fields other than key fields is determined solely by the application.

Key-sequenced files can be accessed sequentially or randomly. An example of sequential access is the generation of a report of the quantity on hand of all parts. Random access would be used to determine the vendor of a particular part.

Figure 2-1 shows the structure of a key-sequenced file.

Relative Files

A relative file consists of a set of records. Each record is uniquely identified among other records in a relative file by a record number, which denotes an ordinal position in a file. The first record in a relative file is designated by record number zero; succeeding records are designated by ascending record numbers in increments of one.

A logical record can vary in size from zero (an empty record) to the maximum record size specified when the file was created. Each record position, however, is always allocated a fixed quantity of storage to ensure that a maximum-length record can be written to any position. (A record's logical length can be changed after the record has been entered.) Each record has a length attribute that can be returned when a record is read.

Records in a relative file can be deleted.

A record occupies a position in a file whether or not the position has been written in or the record has been deleted. For example, for a relative file in which only record number 10 has been written, the FILEINFO procedure's <end-of-file location> parameter would indicate that there are 11 records (including record 0), not just one.

The position where a new record is inserted into a relative file is specified by supplying a record number to the POSITION procedure. Alternatively, you can specify that records be inserted into any available position in a relative file by supplying a record number of "-2D" to POSITION before inserting records into the file. Likewise, you can specify that subsequent records be appended to the end of a file by supplying a record number of "-1D" to the POSITION procedure.

For example, in a relative file in which only record number 10 exists, you can position to an empty location (such as record number 5) and use the WRITE procedure to insert a new record in that location. If you position to record number "-2D", the record is written to some (not necessarily the lowest) empty location. Using the READUPDATE procedure after positioning to an empty location returns file-system error 11 ("record not in file"); the same positioning causes the READ procedure to read the next nonempty record.

When "-2D" or "-1D" is specified for inserting records into a relative file, the actual record number associated with the new record can be obtained through the FILEINFO procedure.

Figure 2-2 shows the structure of a relative file.

Relative files are best suited for applications where random access to fixed-length records is desired and the record number can function as the key to the file. In the earlier inventory example, it would be possible to make the inventory file a relative file where the relative record number was equal to the part number. However, this would probably be wasteful of space since part-numbering schemes typically leave large gaps in the numbers and this would result in many records allocated but not used. On the other hand, an invoice file where the relative record number is equal to the invoice number would be a good application for a relative file, since there are typically no large gaps in this kind of file. Data fields in the record could contain customer number, part number, quantity, price, etc.

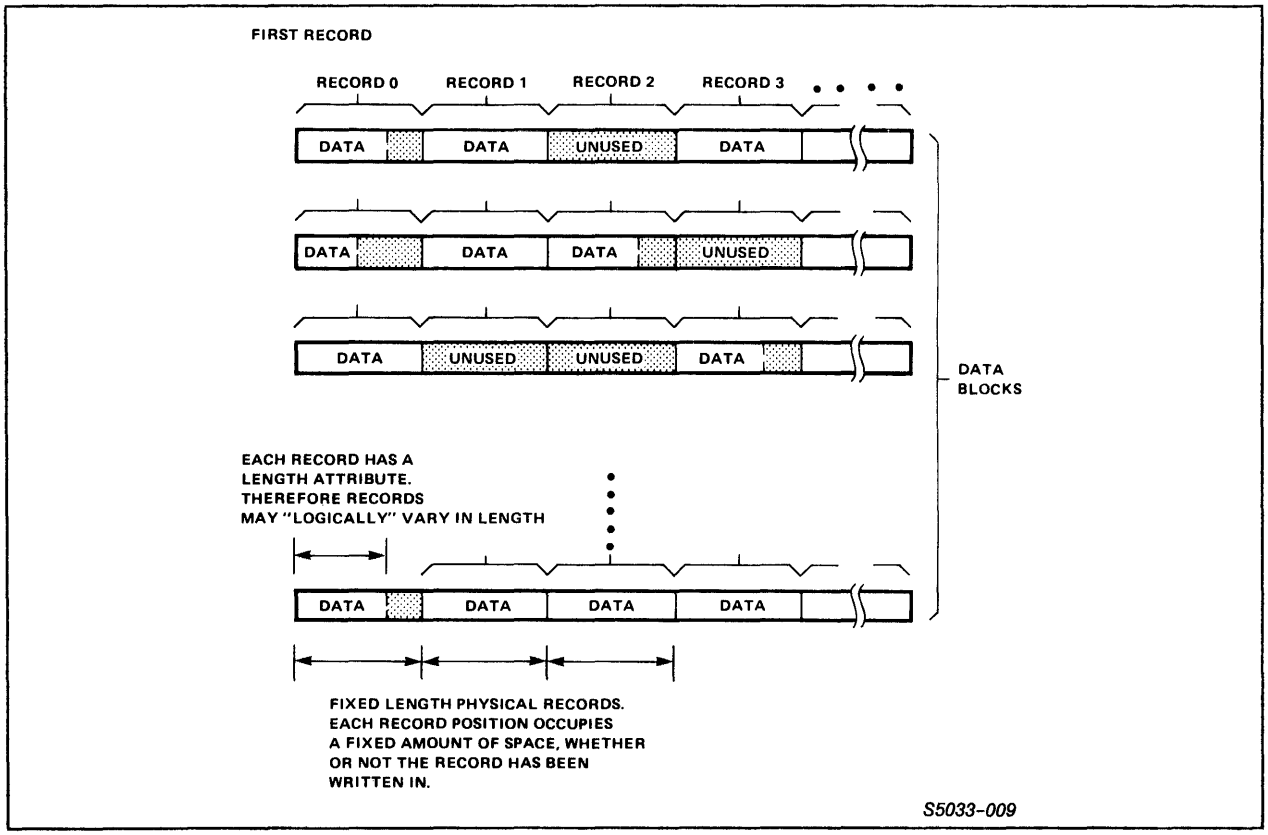


Figure 2-2. Relative File Structure

Entry-Sequenced Files

An entry-sequenced file consists of a set of variable-length records. Each record is uniquely identified among other records in an entry-sequenced file by a record address. Records inserted in an entry-sequenced file are always appended to the end of the file and, therefore, are physically ordered by the sequence presented to the system. To let records be accessed randomly, the FILEINFO procedure returns the record address of an appended record.

A record may vary in length from zero bytes (empty) to the maximum record size specified when the file was created. The number of bytes allocated for a record is the number of bytes written when the record was inserted into the file. Each record has a length attribute that is optionally returned when a record is read. A record's length cannot be changed after the record is written into the file. Records in an entry-sequenced file cannot be deleted.

Figure 2-3 shows the structure of an entry-sequenced file.

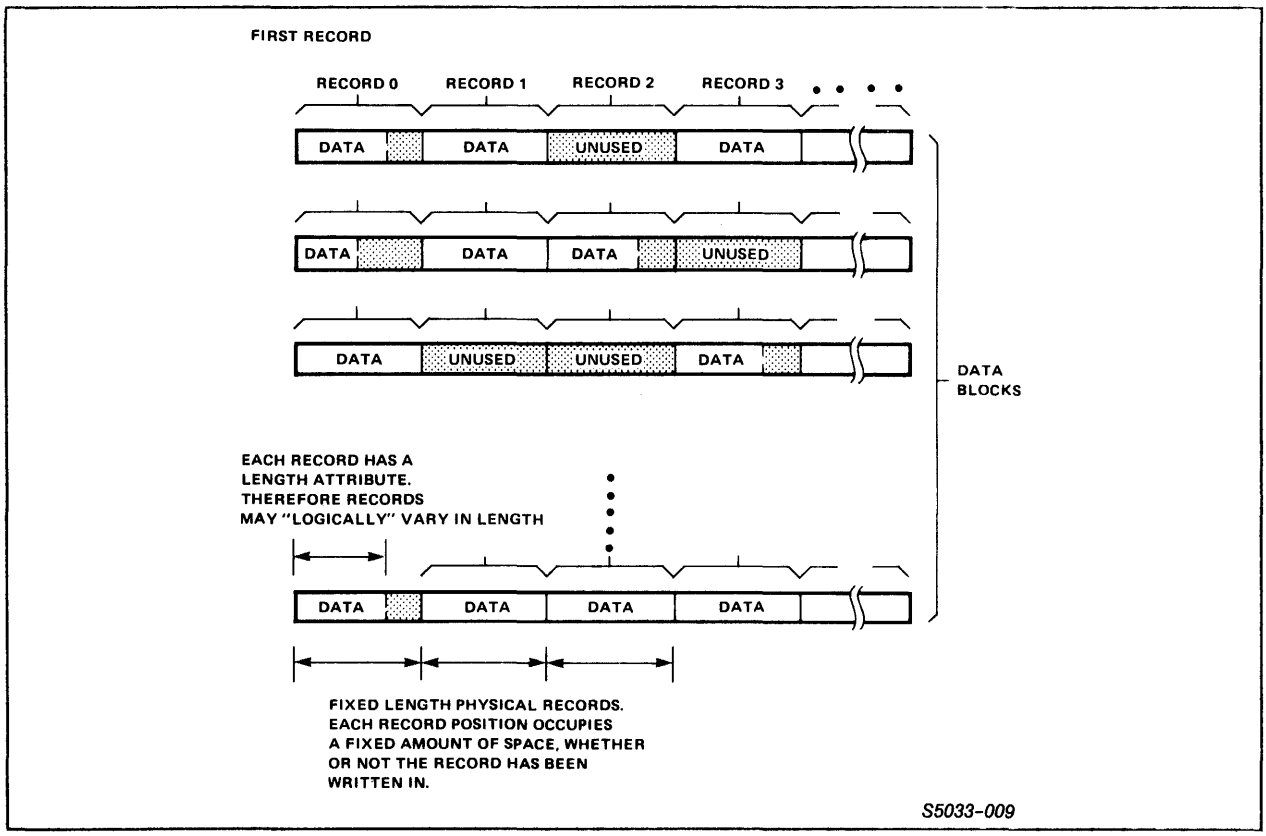


Figure 2-2. Relative File Structure

Entry-Sequenced Files

An entry-sequenced file consists of a set of variable-length records. Each record is uniquely identified among other records in an entry-sequenced file by a record address. Records inserted in an entry-sequenced file are always appended to the end of the file and, therefore, are physically ordered by the sequence presented to the system. To let records be accessed randomly, the FILEINFO procedure returns the record address of an appended record.

A record may vary in length from zero bytes (empty) to the maximum record size specified when the file was created. The number of bytes allocated for a record is the number of bytes written when the record was inserted into the file. Each record has a length attribute that is optionally returned when a record is read. A record's length cannot be changed after the record is written into the file. Records in an entry-sequenced file cannot be deleted.

Figure 2-3 shows the structure of an entry-sequenced file.

Entry-sequenced files are best suited to sequential processing of variable-length data. An example of this type of application is a transaction-logging file. Each transaction becomes a record in the file; the records are stored in the file in the order in which the transactions are made.

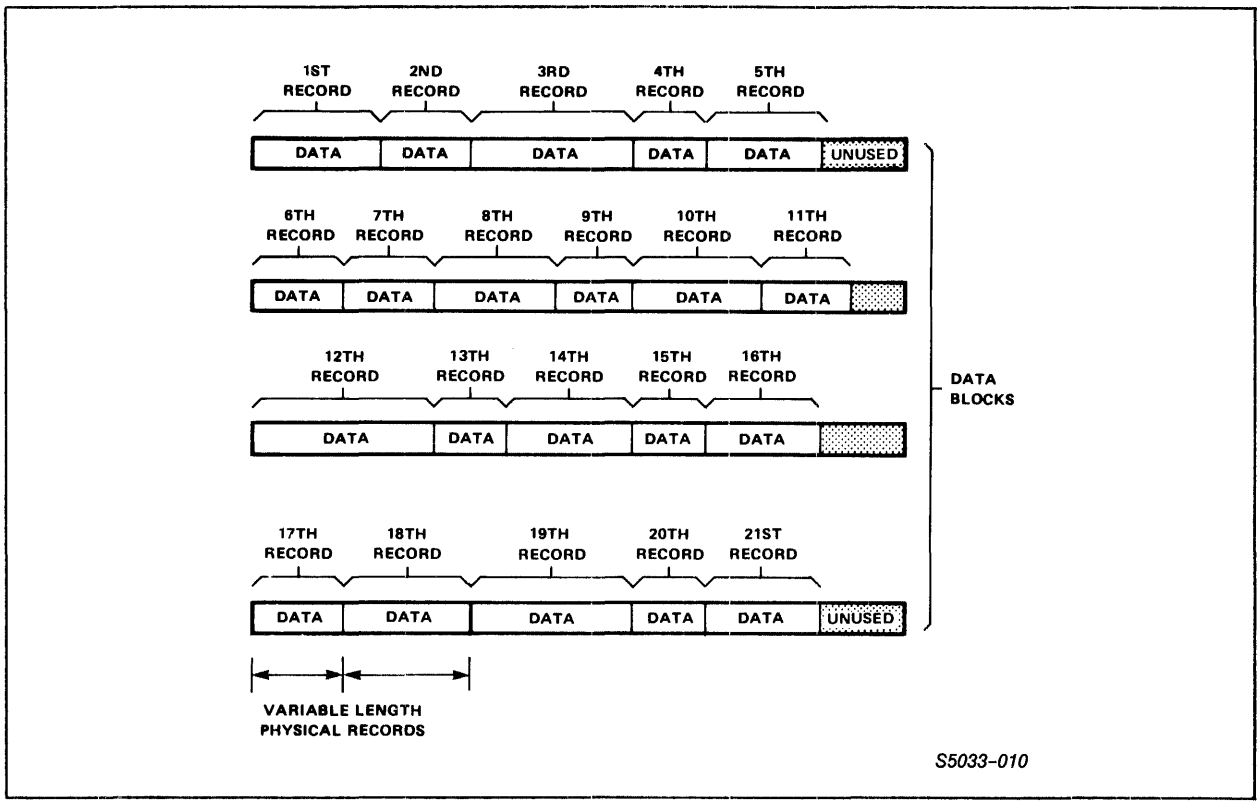


Figure 2-3. Entry-Sequenced File Structure

POSITIONING WITHIN STRUCTURED FILES

This subsection describes how a process can select a subset of a structured file.

Current Key Specifier and Current Access Path

A two-byte key specifier uniquely identifies each key field as an access path for positioning. The key specifier for primary keys is defined as binary zero (ASCII <null><null>). Key specifiers for alternate-key fields are defined by the application and are assigned when the file is created.

The current-key specifier defines the current access path--the order in which the file's records are read sequentially.

The current-key specifier, and therefore the current access path, is implicitly set to the file's primary key when a file is opened or a call is made to the POSITION procedure (for relative and entry-sequenced files only). The access path is set explicitly by calling the KEYPOSITION procedure. Figure 2-4 shows a typical record structure with a primary key and three alternate keys.

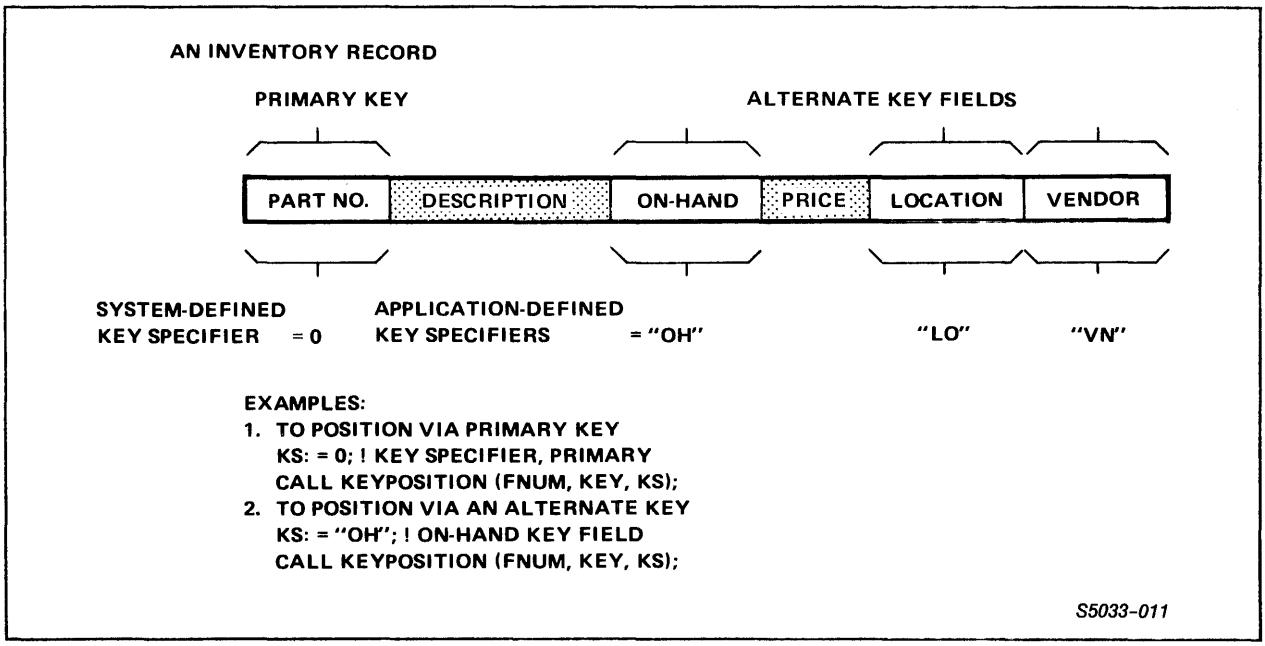


Figure 2-4. Key Fields and Key Specifiers

Current Key Value and Current Position

The current key value defines a subset of records in a file's current access path (see the "Positioning Mode and Comparison Length" subsection below) and sets a file's current position.

You can set current key value explicitly by calling the POSITION or KEYPOSITION procedure. KEYPOSITION sets a position by primary key for key-sequenced files and by alternate key for key-sequenced, relative, and entry-sequenced files. POSITION sets a position by primary key for relative and entry-sequenced files. After a call to READ, the current key value is implicitly set to the key value of the current access path in the record just read.

The current position determines the record to be locked (by a call to LOCKREC) or accessed (by a call to READ[LOCK],

READUPDATE[LOCK], or WRITEUPDATE[UNLOCK]). A record need not exist at the current position. When a file is opened, the current position is that of first record in the file as defined by the file's primary key. Figure 2-5 demonstrates the use of KEYPOSITION in a key-sequenced file.

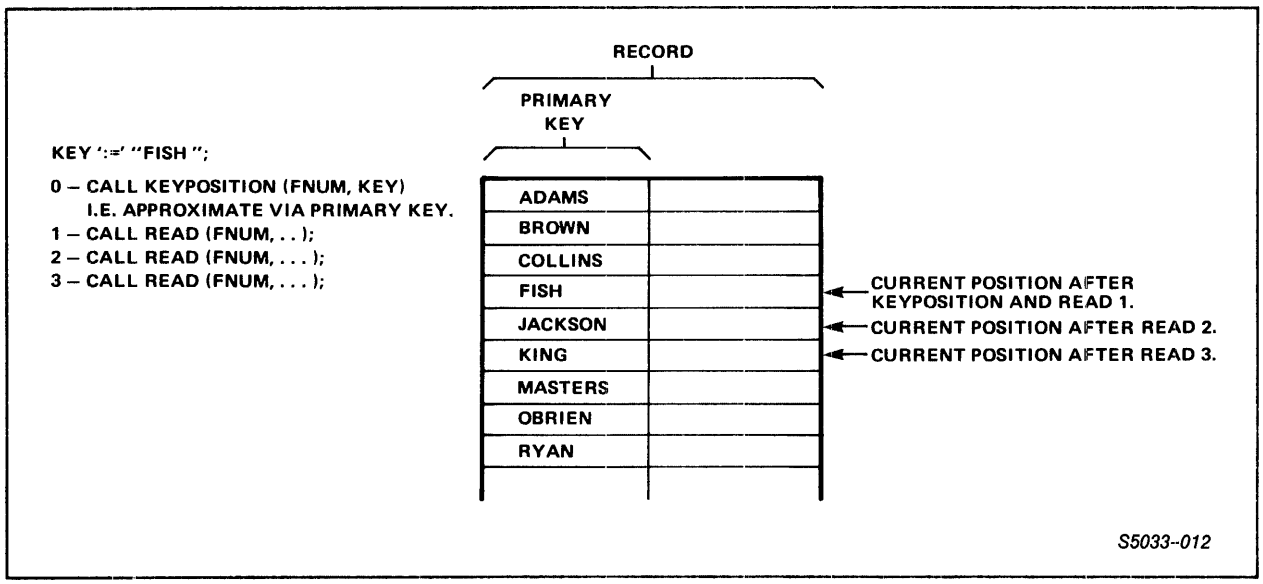


Figure 2-5. Current Position

Positioning Mode and Comparison Length

The positioning mode, comparison length, and current key value determine a subset of records and the first record in that subset to be accessed. The subset of records in the current access path can consist of all, part of, or none of the records in a file.

The positioning mode and comparison length (as well as the current-key specifier and current key value) are set explicitly by the KEYPOSITION procedure and implicitly by the OPEN and POSITION procedures. ENSCRIBE has three positioning modes: approximate, generic, and exact.

APPROXIMATE. Approximate positioning means the first record accessed is the one whose key field, as indicated by the current-key specifier, contains a value equal to or greater than the current key value. After approximate positioning, sequential READ operations to the file return ascending records until the last record in the file is read; an end-of-file indication is then returned. When a file is opened, the positioning mode is set to approximate and the comparison length is set to 0.

After approximate positioning, sequential READ operations to a relative file will skip non-existent records.

GENERIC. Generic positioning means the first record accessed is the one whose key field, as designated by the current-key specifier, contains a value equal to the current key value for the number of bytes specified by the comparison length. After generic positioning, sequential READ operations to the file return ascending records whose key matches the current key value (for the comparison length). When the current key no longer matches, an end-of-file indication is returned.

For relative and entry-sequenced files, generic positioning by the primary key is equivalent to exact positioning.

EXACT. Exact positioning means the only records accessed are those whose key field, as designated by the current-key specifier, contains a value that is (1) exactly as long as the specified comparison length and (2) equal to the current key value. When the current key no longer matches, an end-of-file indication is returned. If the key field has a unique value, exact positioning accesses no more than one record.

ALTERNATE KEYS

For each file having one or more alternate keys, at least one alternate-key file exists. Each record in an alternate-key file consists of:

- Two bytes for the <key specifier>
- The alternate-key value
- The primary-key value of the associated record in the primary file

The length of an alternate-key record is at least

2 + alternate-key field length + primary-key length

Figure 2-6 shows how alternate keys are implemented.

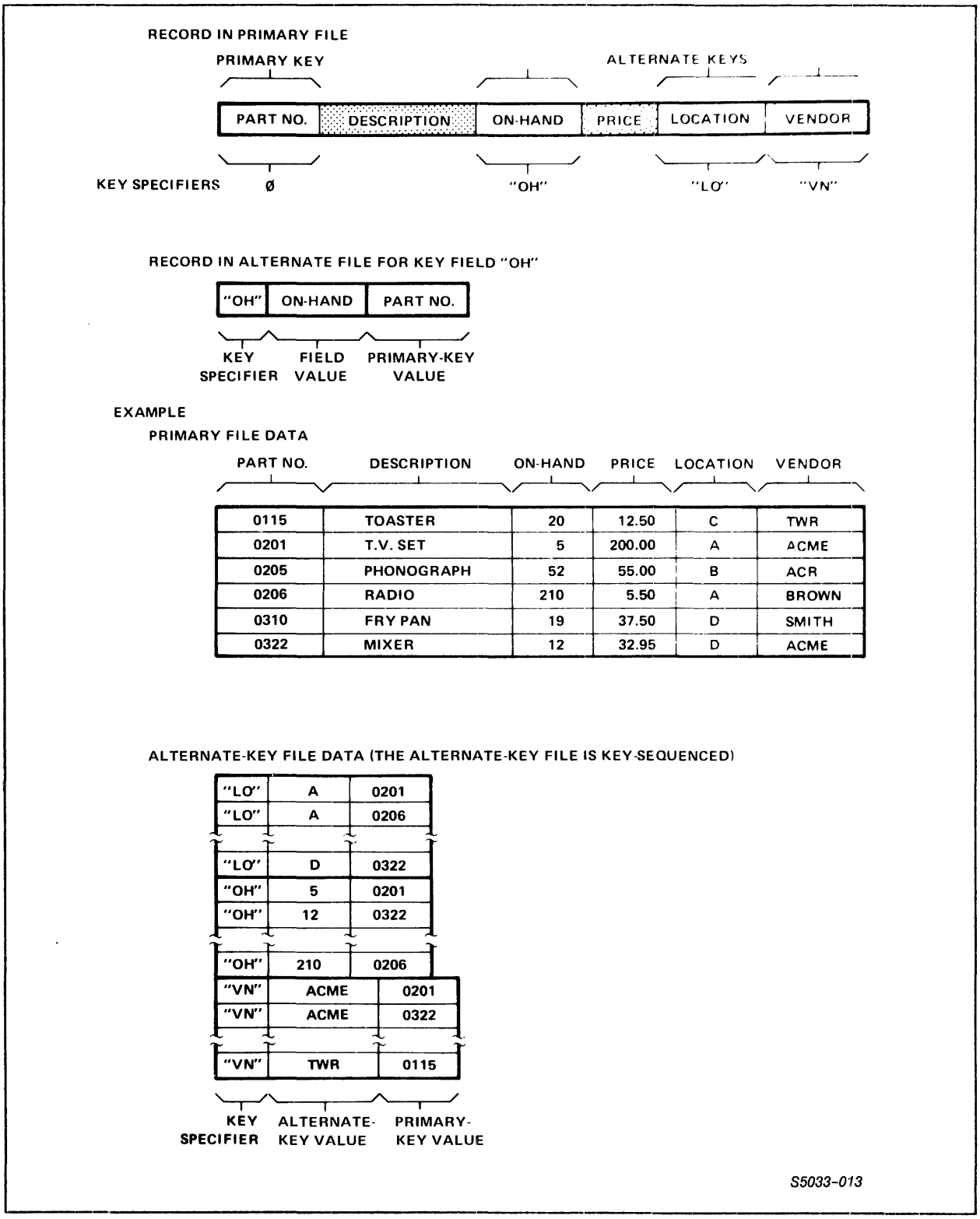


Figure 2-6. Alternate-Key Implementation

Alternate-Key Attributes

When an alternate key is defined, these attributes can be assigned:

- Null value
- Unique alternate key
- No automatic update.

A null value is a byte value that, when encountered in all positions of the indicated key field during a record insertion, causes the alternate-key file entry for the field to be omitted. Any records containing only the null value in the alternate-key field are skipped when the file is read sequentially via an alternate-key field having a null value defined.

If an alternate-key field is defined as requiring a unique value, any attempt to insert a record having a duplicate key value in that field is rejected with file-system error 10, "record already exists."

The data-base designer can specify that an alternate-key field not be updated by the system when a change to that field occurs.

Alternate Keys in a Key-Sequenced File

An example of alternate-key usage in a key-sequenced file would be a file whose records consist of the vendor name and the part number. The primary key to this file would be the part number (it could not be the vendor name, because that is not unique). To produce a report listing of all parts supplied by a given vendor, generic positioning would be done via the desired vendor. Then the file would be read sequentially until the vendor name field is not equal to the desired vendor (at which time the system will return an end-of-file indication). The records associated with a given vendor would be returned in ascending order of the part number.

Alternate Keys in a Relative File

An example of alternate-key usage in a relative file would be a file of employee data. The primary key (a record number) would be an employee number. One alternate-key field would be an employee name.

Alternate Keys in an Entry-Sequenced File

An example of alternate-key usage in an entry-sequenced file would be in a transaction-logging file. The primary key (a record address) would indicate the order in which transactions occurred. An alternate-key field might show the terminal in the system that initiated a transaction. To list all transactions for given terminal in the order in which they occurred, generic positioning would be done using the field value of the desired terminal, then the file would be read sequentially.

COMPARISON OF STRUCTURED-FILE CHARACTERISTICS

Table 2-1 compares the characteristics of the three structured file types.

Table 2-1. Comparison of Key-Sequenced, Relative, and Entry-Sequenced Files

Key-Sequenced	Relative	Entry-Sequenced
Records are ordered by value in primary-key field.	Records are ordered by relative record number.	Records are in the order in which they are entered.
Access is by primary or alternate key.	Access is by record number or alternate key.	Access is by record address or alternate key.
Space occupied by a record depends on length specified when written.	Space allowed per record is specified when the file is created.	Space occupied by a record depends on length specified when written
Free space in block or at end of file is used for adding records.	Empty positions in file are used for adding records.	Space at end of file is used for adding records.
Records can be deleted, shortened, or lengthened. (within the maximum size specified).	Records can be deleted, shortened, or lengthened (within the maximum size specified).	Records cannot be deleted, shortened, or lengthened.

Table 2-1. Comparison of Key-Sequenced, Relative, and Entry-Sequenced Files (continued)

Key-Sequenced	Relative	Entry-Sequenced
Space freed by deleting or shortening a record is reused within its block.	Space freed by deleting a record can be reused.	A record cannot be deleted, but its space can be used for another record of the same size.

SECTION 3

USE OF PROCEDURE CALLS

Application processes have access to ENSCRIBE disc files through a set of file-system procedures or through a separate set of sequential I/O procedures. These two sets of procedures should not be used together.

This section introduces those procedures that affect ENSCRIBE files.

FILE-SYSTEM PROCEDURES

This subsection contains a brief summary of the file-system procedures and a discussion of their common characteristics. Detailed syntax descriptions of all file-system procedures and parameters are included in the System Procedure Calls Reference Manual.

File-System Procedures Summary

Table 3-1, on the next three pages, summarizes the functions of file-system procedures.

Table 3-1. File-System Procedures

Procedure	Function
AWAITIO	waits for completion of an outstanding I/O operation pending on an open file.
CANCELREQ	cancels the oldest outstanding operation, optionally identified by a <tag>, on an open file.
CLOSE	stops access to an open file and purges a temporary disc file.
CONTROL	executes device-dependent operations to an open file.
CREATE	creates a new disc file (permanent or temporary).
DEVICEINFO	provides the device type and physical record size for a file (open or closed). The <device type> codes are described in the System Procedure Calls Reference Manual.
DEVICEINFO2 (NonStop systems only)	provides the device type and physical record size for a file (open or closed) and determines whether the volume is formatted as a DP1 or DP2 volume. The <device type> codes are described in the System Procedure Calls Reference Manual.
FILEERROR	helps determine whether a failed call should be retried.
FILEINFO	provides error and characteristic information about a file.
FILERECINFO	provides characteristic information about an open disc file.
FNAMECOLLAPSE	collapses an internal file identifier to external form.
FNAMECOMPARE	compares two internal file identifiers to determine whether they refer to the same file or device.

Table 3-1. File-System Procedures (continued)

Procedure	Function
FNAMEEXPAND	expands an external file identifier to internal form.
GETDEVNAME	returns the $<device name> or $<volume name> associated with a logical device number if such a device exists; otherwise, the name of the next-higher logical device number is returned.
KEYPOSITION	sets position by primary key within a key-sequenced file or by alternate key within any structured file; defines a subset of the file for subsequent access, by setting the current position, access path, and positioning mode.
LOCKFILE	locks an open disc file, making the file inaccessible to other accessors.
LOCKREC	locks a record in an open disc file so that other processes cannot access the record.
NEXTFILENAME	returns the next disc-file name in alphabetical sequence following the designated file name.
OPEN	establishes communication with a file.
POSITION	sets position by primary key within a relative or entry-sequenced file; defines a subset of the file for subsequent access, by setting the current position, access path, and positioning mode; also can specify new current position in an unstructured file.
PURGE	erases a closed disc file from the system.
READ	returns the record indicated by the value of the current key; READ is used when sequentially reading an open file.
READLOCK	is the same as READ but locks the record before reading it.

Table 3-1. File-System Procedures (continued)

Procedure	Function
READUPDATE	returns the record indicated by the current key value; READUPDATE is used to randomly read an open file.
READUPDATELOCK	is the same as READUPDATE except that it locks the record before reading it.
REFRESH	writes information (such as the end-of-file pointer), contained in file control blocks (FCBs) in main memory, to the associated physical disc volume.
RENAME	renames an open disc file and makes a temporary disc file permanent.
REPOSITION	restores the disc-file position information saved with a previous SAVEPOSITION call.
SAVEPOSITION	saves the current disc-file position information; a later call to REPOSITION restores the saved position.
SETMODE	sets device-dependent functions in an open file.
SETMODENOWAIT	sets device-dependent functions in a "no-wait" manner for an open file.
UNLOCKFILE	unlocks an open disc file currently locked by the caller. UNLOCKFILE also unlocks any records in the designated file that are currently locked by the caller.
UNLOCKREC	unlocks a record currently locked by the caller, so other processes can access the record.
WRITE	inserts (adds) a new record into an open disc-file location read by the last call to READ or READUPDATE.

Table 3-1. File-System Procedures (continued)

Procedure	Function
WRITEUPDATE	replaces (updates) or deletes data in the existing record indicated by an open file's current key value.
WRITEUPDATEUNLOCK	is the same as WRITEUPDATE, but unlocks the record after it is updated or deleted.

Characteristics of ENSCRIBE Procedure Calls

This section describes features common to all ENSCRIBE file-system procedure calls.

COMPLETION. If a file is open with "no-wait" I/O specified, calls to CONTROL, LOCKFILE, LOCKREC, READ, READLOCK, READUPDATE, READUPDATELOCK, UNLOCKFILE, UNLOCKREC, WRITE, WRITEUPDATEUNLOCK, or SETMODENOWAIT must be completed by a corresponding call to AWAITIO.

If a file is open with "no-wait" I/O specified, calls to KEYPOSITION, POSITION, RENAME, REPOSITION, SETMODE, or SETMODENOWAIT are rejected with file-system error 27 if there are any outstanding (uncompleted) operations pending.

Regardless of whether the file was opened with "wait" or "no-wait" I/O specified, a return from a call to CANCELREQ, CLOSE, CREATE, DEVICEINFO, DEVICEINFO2, FILEINFO, FILERECINFO, KEYPOSITION, NEXTFILENAME, OPEN (unless flag <8> is set to 1), POSITION, PURGE, RENAME, or SETMODE indicates a completion.

FILE-NUMBER PARAMETERS. All file-system procedures except DEVICEINFO, DEVICEINFO2, CREATE, OPEN, NEXTFILENAME, REFRESH, and PURGE use the <file number> parameter, returned from the OPEN procedure, to identify which file the call refers to. The DEVICEINFO, DEVICEINFO2, CREATE, OPEN, and PURGE procedures refer to the file by its file identifier; the LASTRECEIVE and REPLY procedures always refer to the $RECEIVE file (i.e., interprocess communication). For every procedure that has a <file number> parameter, except OPEN and AWAITIO, the file number is an INT:value parameter.

TAG PARAMETERS. An application-specified double integer (INT(32)) tag can be passed as a calling parameter when an I/O operation (read or write) is initiated with a "no-wait" file. The tag is passed back to the application process, through the AWAITIO procedure, when the I/O operation completes. The tag is useful for identifying individual file operations and can be used in application-dependent error-recovery routines.

BUFFER PARAMETER. In an application program used to transfer data between the application process and the file system, the data buffers must be integer (INT) or double-integer (INT(32)) and must reside in the program's data area ('P' relative read-only arrays are not permitted).

TRANSFER-COUNT PARAMETER. The <transfer count> parameter of any file-system procedure refers to the number of bytes to be transferred. From 0 to 4096 bytes can be transferred in a single operation with an ENSCRIBE disc file. The actual maximum transfer count may be less than 4096, due to the amount of buffer space assigned to the disc during system generation (SYSGEN). (The amount of buffer space configured for a disc volume can be obtained via the DEVICEINFO procedure.)

CONDITION CODES. All file-system procedures return a condition code indicating the outcome of the operation. THE CONDITION CODE SHOULD ALWAYS BE CHECKED AFTER A CALL TO A FILE SYSTEM PROCEDURE and should be checked before an arithmetic operation is performed or a value is assigned to a variable. Generally, the condition codes have these meanings:

<	(CCL)	An error occurred (call the file-system FILEINFO procedure to determine the error).
>	(CCG)	A warning message was generated (typically end-of-file, but see the individual procedures for the meaning of CCG or call FILEINFO to obtain an error number).
=	(CCE)	The operation was successful.

ERRORS. An error number is associated with each call completion. The error numbers fall into three major categories, as shown in Table 3-2. The setting of the condition code indicates the category of the error associated with a completed call.

The error number associated with an operation on an open file can be obtained by calling the FILEINFO procedure and passing the file number of the file in error, as in this example:

```
CALL FILEINFO(in^file, err^num);
```

The error number of a preceding AWAITIO on any file or of a waited OPEN that failed can be obtained by calling FILEINFO with file number -1, as in this example:

```
CALL FILEINFO(-1, err^num);
```

Note: If the OPEN procedure fails, it returns -1 to the <file number> parameter.

Similarly, the error number of a preceding CREATE or PURGE operation that failed can be obtained by calling FILEINFO with file number -1.

Section 5 describes error recovery in detail. The System Messages Manual for your system contains a complete list of the error numbers and their meanings.

Table 3-2. Error-Number Categories

Error	CC	Category
0	CCE	No error. The operation was executed successfully.
1-9	CCG	Warning. The operation was executed, with the indicated exception condition. For message 6 ("system message received"), data are returned in the application process's buffer.
10-255	CCL	Error. The operation encountered an error. For data-transfer operations, either none or part of the specified data were transferred (except data communication error 165, which indicates normal completion with data returned in the application process's buffer).
300-511	CCL	Error. These errors are reserved for process application-dependent use.

CHECKING ACCESS MODE AND SECURITY. The disc file must be open with read or read/write access for a READ, READLOCK, READUPDATE, or READUPDATELOCK call to be successful. Otherwise, the call will be rejected with file-system error 49, "access violation."

The disc file must be open with write or read/write access for CONTROL, WRITE, WRITEUPDATE, or WRITEUPDATEUNLOCK calls to be successful (otherwise the call will be rejected with file-system error 49, "access violation."

The caller must have purge access to a disc file for PURGE or RENAME calls to be successful (otherwise the call will be rejected with file-system error 48, "security violation."

External Declarations

Like all other procedures in an application program, the file-system procedures must be declared before being called. These procedures are declared as having "external" bodies. The external declarations for these procedures are provided in system file $SYSTEM.SYSTEM.EXTDECS. The compiler command SOURCE, specifying this file, should be included in the source program after the global declarations but before the first call to one of these procedures, as in this example:

```
<global declarations>
?SOURCE $SYSTEM.SYSTEM.EXTDECS (<external procedure name>, ...)
<procedure declarations>
```

Each external procedure used in the program should be specified in the SOURCE command. For example,

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS ( OPEN, READ, WRITE, POSITION,
?                                KEYPOSITION, WRITEUPDATE, CLOSE )
```

compiles only the external declarations for the OPEN, READ, WRITE, POSITION, KEYPOSITION, WRITEUPDATE, and CLOSE procedures.

SEQUENTIAL I/O PROCEDURES (SIO)

The sequential I/O (SIO) procedures provide TAL programmers with a standardized set of procedures for performing common input and output operations. These operations include reading and writing IN and OUT files, and handling BREAK from a terminal. These procedures are primarily intended for TANDEM subsystem and user utility programs. The primary benefit is that programs using these procedures can treat different file types in a consistent and predictable manner.

The sequential I/O procedures and their functions are listed in Table 3-3.

Table 3-3. SIO Procedures

Procedure	Function
CHECK^BREAK	checks whether the break key was pressed
CHECK^FILE	retrieves file characteristics
CLOSE^FILE	closes a file
GIVE^BREAK	disables the break key
OPEN^FILE	opens a file for access by the SIO procedures
READ^FILE	reads from a file
SET^FILE	sets or alters file characteristics
TAKE^BREAK	enables the break key
WAIT^FILE	waits for the completion of an outstanding I/O operation
WRITE^FILE	writes to a file

The SIO procedures also contain a set of DEFINEs and LITERALs that allocate control-block space, specify file-opening characteristics, set file-transfer characteristics, and check file-transfer characteristics.

Some characteristics of the SIO procedures are:

- All file types are accessed in a uniform manner. File access characteristics, such as access mode, exclusion modes, and record size, are selected according to device type and the intended access. The SIO procedures' default characteristics are set to facilitate their most general use.

- Error recovery is automatic. Each fatal error causes the display of a comprehensive error message, all files to close and the process to abort. Both the automatic error-handling and the display of error messages can be turned off, so the program can do the error-handling.

- The OPEN^FILE procedure lets an application alter the characteristics of SIO operations when a file is opened. Also, the SET^FILE procedure makes this possible before or after the file is opened. Some optional characteristics are:

--record blocking and deblocking

--duplicative file capability where data read from one file are automatically echoed to another file

--an error-reporting file, where all error messages are directed. When a particular file is not specified, the error-reporting file is the home terminal.

- The SIO procedures can be used with the INITIALIZER procedure to make run-time changes. File-transfer characteristics, such as record length, can be changed using the GUARDIAN Operating System Command Interpreter's ASSIGN command.

- The SIO procedures retain information about the files in file control blocks (FCBs). There is one FCB for each open file plus one common FCB which is linked to the other FCBs.

A thorough discussion of the SIO procedures is in the GUARDIAN Operating System Programmer's Guide.

SECTION 4

FILE CREATION

Files on a disc must be created before being opened for access. This section discusses the concepts necessary for choosing appropriate parameters when creating a disc file.

An ENSCRIBE file is created through the File Utility Program (FUP) or through a programmatic call to the CREATE procedure. The Data Definition Language (DDL) compiler can automatically generate a FUP command file that can be used to create files described by the DDL source schema.

All partitions of a multivolume file are automatically created when the first partition is created.

For each structured file having one or more alternate-key fields, the user must create the associated alternate-key files. An alternate-key file is created as a key-sequenced file.

When you create an ENSCRIBE disc file, you must consider:

- File type
- File code
- Extents
- Logical records
- Blocks
- Characteristics of key-sequenced files
- Characteristics of structured files having alternate keys
- Characteristics of partitioned (multivolume) files

Also, creation of a DP2 file offers several options not available for a DP1 file, including:

- Maximum number of file extents
- Audit compression
- Buffering
- Verification

FILE UTILITY PROGRAM (FUP)

Typically, ENSCRIBE files are created through use of the File Utility Program (FUP). The FUP commands related to file creation are summarized in Table 4-1.

Table 4-1. FUP Commands Related to File Creation

Command	Function
SET	establishes one or more creation-parameter values for a subsequent creation. Parameter values can be specified explicitly and can be set to match those of an existing file.
SHOW	displays current settings of the creation parameter values.
CREATE	creates a file using the current creation-parameter values. The current parameter values can be overridden in the CREATE command (without affecting the current settings) by specifying alternate values for designated parameters.
RESET	sets one or more creation-parameter values to their default settings.
INFO	displays file characteristics of one or more files.

For creating disc files, FUP is intended to be used in this manner:

1. The user sets the creation parameters, describing the file to be created, by executing one or more SET commands. The SET command can set the creation parameters to match those of an existing file; this is useful for duplicating file structures.

2. The user verifies the settings of the creation parameters by executing a SHOW command. Necessary changes can be made to parameter values by using the SET command again.

3. Once the creation parameters have the desired settings, the file is created by means of the CREATE command. An option to the CREATE command permits the user to override current creation-parameter settings.

4. After file creation, the current settings are still in force. Other files can be created using these settings.

FUP resides in volume $SYSTEM. Normally, it is run through use of the GUARDIAN Operating System Command Interpreter. For a complete description of all the FUP commands, see the GUARDIAN Operating System Utilities Reference Manual. Also see the FUP discussion in the GUARDIAN Operating System User's Guide.

CREATE PROCEDURE

The CREATE procedure defines a new structured or unstructured disc file. The file can be either temporary (and deleted when closed) or permanent. If a temporary file is created, CREATE returns a file name suitable for passing to the OPEN procedure.

The syntax of the CREATE procedure call is described in detail in the System Procedure Calls Reference Manual.

CONSIDERATIONS FOR STRUCTURED AND UNSTRUCTURED FILES

This subsection describes file-creation parameters common to all four file types: key-sequenced, relative, entry-sequenced, and unstructured.

File Types

KEY-SEQUENCED FILES. A key-sequenced file consists of a set of variable-length records, each of which is uniquely identified by the value of its primary-key field. Records in a key-sequenced file are logically stored in order of ascending primary-key values.

RELATIVE FILES. A relative file consists of a set of variable-length records, each of which is uniquely identified by a record number. A record number denotes an ordinal position in a file. The first record in a relative file is designated by record number zero; succeeding records are designated by ascending record numbers in increments of one. A record occupies a position in a file whether or not the position has been written to.

ENTRY-SEQUENCED FILES. An entry-sequenced file consists of a set of variable-length records. Records inserted in an entry-sequenced file are always appended to the end of the file and, therefore, are physically ordered by the sequence in which they are presented to the system.

UNSTRUCTURED FILES. An unstructured disc file is essentially a byte array, starting at byte zero and continuing to the last byte of the file, marked by the end-of-file pointer. The file system imposes no structure on an unstructured disc file. Grouping of data into records and where records are located within the file are the responsibility of the application process.

Unstructured files are accessed on the basis of a relative byte address. The current location in a file is automatically incremented by the file system after a READ or a WRITE. The application process can position to a specific location within the file by supplying the file system a relative byte address with the POSITION procedure. After a call to POSITION, the application process can issue any other file-system call (READ, WRITE, READUPDATE, etc.).

Partitioned (Multivolume) Files

A file may be partitioned into as many as 16 disjoint subfiles, each residing on a different volume. Moreover, each volume can be on a different node in a network.

ADVANTAGES. After a partitioned file is created, the fact that it resides on more than one volume (and perhaps on more than one node) is transparent to the application program. The entire file is opened for access by supplying the name of the primary partition to OPEN. Attempts to separately open secondary partitions of the file are rejected, except when using unstructured access, which deals with only one partition (see the OPEN procedure in the System Procedure Calls Reference Manual).

Partitioned files can be valuable for a number of reasons. The most obvious one is that a file can be created whose size is not limited by the size of a physical disc pack. In addition, by spreading a file across more that one volume, the concurrency of access to records in the file can be increased.

If the file is located on multiple volumes on the same controller, the operating system will take advantage of the controller's overlapped seek capability (that is, many drives can be seeking while one is transferring). If the file spans volumes connected to different controllers on the same processor, overlapping transfers will occur (up to the limit of the I/O

channel's bandwidth). And if the file resides on volumes connected to controllers on different processors, the system will perform overlapped processing of requests and overlapped transfers not limited by the bandwidth of a single I/O channel.

Partitioned files can also accommodate more locks, because the locking limit applies to each partition rather than the whole file. (Locking limits are discussed in Section 5, "File Access".)

Partitioned file records can also reside in multiple caches, which can result in fewer disc accesses.

FILE IDENTIFIERS. Each partition has a directory entry on the volume on which it resides. The file names for all partitions are identical except for the difference of the volume name.

FEW DIFFERENCES AMONG PARTITIONS. All partions of a file must be on either DP1 or DP2 volumes, and all partitions must be either audited or unaudited. They cannot be mixed.

Primary and secondary extent sizes can differ from one partition to another within the same partitioned file. This and the volume names are the only file attributes on which partitions can differ.

PARTIAL-KEY VALUE. For key-sequenced files, a partial-key value of from one to <key length> bytes is supplied for each partition. All records with keys greater or equal to the partition's key, but less than the next partition's key, are assigned to the partition. The partial key for the primary partition is a string of all nulls.

For file types other than key-sequenced, the size of a partition and its order in the partition-parameter list determines which records are located in that partition.

Block Sizes and Extents

The sizes of a DP2 key-sequenced file's index blocks (IBLOCK in FUP's SET command) and data blocks (BLOCK) must be identical. When creating a file, the DP2 disc process ignores the specified IBLOCK size and uses the size specified in BLOCK.

If a DP1 file's index-block size and data-block size differ, then no extent size should be smaller than the larger of the two block sizes. If this rule is ignored, multiple extents are allocated

every time a block is acquired and some operations (such as FUP's LOAD command on a key-sequenced file) return file-system error 21 ("illegal <count> specified").

The block size of a key-sequenced file should be large, relative to the record size and especially relative to the key size, to reduce the number of block splits as records are inserted into the file. Also, a larger data block implies more data records per block and therefore fewer index records and fewer index blocks.

The extent size of a DP2 file must be an integral multiple of the file's buffer size (for an unstructured file) or block size (for a structured file), which in turn must be one of 512, 1024, 2048, or 4096. When the file is created, the file system rounds up the extent size to enforce this requirement, if necessary.

The file system also rounds up any extent size specified as an odd number of pages (one page = 2048 bytes) when the block size (of a structured file) or buffer size (of an unstructured file) is 4096. Therefore, if you want to have a file with one-page or three-page extents, the buffer size or block size must be set to 2048 or less when the file is created.

If a DP1 file's block size is 4096, the DP1 disc process rounds an extent size of 1 page up to 2 pages because no block of a structured file can span more than 2 extents. DP1 does not round up other extent sizes, including the odd ones larger than 1 page.

File Codes

An application-defined file code can be assigned to a disc file when the file is created. The file code is typically used to categorize files by the information they contain. File codes 100 through 999 are reserved. Use of file codes with this reserved range may cause unpredictable results such as system failures or erroneous results.

CONSIDERATIONS FOR STRUCTURED FILES

This section describes ENSCRIBE facilities that apply to structured files only (key-sequenced, relative, and entry-sequenced files).

Logical Records

A logical record is the unit of information transferred between an application program and the file system. For each file, a maximum logical record length must be specified. If an application program attempts to insert a record longer than the specified record length, the insert operation is rejected with an "illegal count" error.

Record length is determined by the needs of the application, within these guidelines:

- Records in a file can be of varying lengths but cannot be larger than the <record length> defined when the file is created.
- The maximum possible record size is determined by the data-block size and the type of file.

For relative and entry-sequenced files, the maximum record length cannot exceed <data-block size> - 24. This means, for relative and entry-sequenced files with a data-block size of 4096 (the maximum), the maximum record length is 4072.

For key-sequenced files with the DP1 disc process, the maximum record length is 1/2 * (<data-block size> - 26); with a data-block size of 4096, this means the maximum record length is 2035 bytes.

Under the DP2 disc process, the record length can be as long as (<block size> - 34); thus, with a block size of 4096, the maximum record length is 4062 bytes.

Blocks

The block is the unit of information transferred between the file system and a disc volume. A block consists of one or more logical records and associated control information. This control information, which is used only by the system, differs between DP1 files and DP2 files. Both block formats are shown in Appendix B.

These points should be considered when choosing a structured file's block size:

- The block size must be a multiple of sector size (512 bytes) and cannot be greater than 4096 bytes. For DP2 files, the block size is further restricted to 512, 1024, 2048, or 4096 bytes.

- Regardless of the record length, the maximum number of records that can be stored in a single block is 511.

- Block size under the DP1 disc process must include 22 bytes for block control and two bytes per record for record control. Therefore, the number of records per block is

  N = (B - 22) / (R + 2)

  where B is <block size> and R is <record length>. If records are of varying lengths, then N is the average number of records per block and R is given as the average record length.

  Under the DP2 disc process, the block-control allowance is 32 for key-sequenced files and 22 for entry-sequenced and relative files. Therefore, the number of records per block is

  N = (B - 32) / (R + 2)

  for key-sequenced files and

  N = (B - 22) / (R + 2)

  for entry-sequenced and relative files.

  Records cannot span blocks. Therefore, the block size must be at least

  <record length> + 2 + 32

  for a key-sequenced file under the DP2 disc process or

  <record length> + 2 + 22

  for any other file.

For key-sequenced files, the block size (under the DP2 disc process) or data-block size (under the DP1 disc process) determines the maximum record length that can be defined for the file. The maximum lengths for DP1 and DP2 records are given in the "Logical Records" subsection above.

Considerations for Key-Sequenced Files

Considerations for key-sequenced files include compression, primary keys, and index blocks.

COMPRESSION AND COMPACTION. Data files and index files can be compressed for more space-efficient storage. Compression, sometimes called front compression, often reduces disc-head movement because more data can fit in a smaller area, but it can also incur some extra system overhead.

ENSCRIBE compresses data records by eliminating leading characters that are duplicated from one record to the next and replacing them with a one-byte count of the duplicate characters. For example, if these three records are inserted in a file after FRANKLIN, BEN:

```
JONES, JANE
JONES, JOHN
JONES, SAM
```

what is actually written on the disc is:

```
0JONES, JANE
8OHN
7SAM
↑
count of duplicate characters.
```

because the second record's first eight bytes are identical with those of the first record and the third record's first seven bytes are identical with those of the second record.

When a file is created by the File Utility Program (FUP), the SET command's DCOMPRESS, ICOMPRESS, and COMPRESS parameters establish whether compression will be applied to the data file, its index file, or both. When a file is created by the CREATE procedure, bits 11 and 12 of the <file type> parameter establish compression.

When deciding whether compression should be used, you should consider these facts:

- If compression is used, one additional byte per record may be required in each block. Moreover, additional system processing is required to expand the compressed records.
- If data compression is used, the record's primary-key field must begin at offset [0] of the record. Therefore, variable-length primary keys cannot be used unless the entire record is the primary-key field.
- If there is considerable similarity among the record's primary-key values then data compression is desirable. If not, then compression just adds unnecessary system overhead.

- If there is enough similarity among records that the first records of successive blocks have similar primary-key values, then index compression is desirable.

- Data compression is useful for alternate-key files where several alternate keys tend to have the same value.

A similar mechanism is used to compact all index records, regardless of whether compression is specified. However, index compaction eliminates the trailing part of similar records where compression eliminates the leading part.

The file system uses one index record per data block, rather than one per data record. This index record is formed by comparing the first primary key of the block with the last primary key of the previous block. If these keys are identical for the first N bytes, then the first N + 1 bytes of the block-starting key are used for the index record. For example, with these four data blocks:

```
ALLEN, HARRY      FRASER, IAN      JONES, JOHN      LARIMER, JO
...               ...              ...              ...
FRANKLIN, BEN     JONES, JANE      LANSON, SAM      MARNER, SID
```

these three index records are actually written to disc:

```
FRAS
JONES, JO
LAR
```

The primary-key field in an alternate-key file's data record is not compacted. However, since the alternate-key file is itself a key-sequenced file, the index records for the alternate-key file are compacted.

With the DP2 disc process, you can apply audit-checkpoint compression to any file that is audited or opened with <sync depth> greater than 0. This is decribed in detail under "Audit-Checkpoint Compression" in Section 5.

PRIMARY KEY. For a key-sequenced file, the offset in the record where the primary-key field begins and the length of key field must be specified.

Some considerations when choosing the offset of the primary-key field:

- The primary-key field can begin at any offset within a record and can be of any length up to

  $min(<record length> minus <offset>, 255)

- If data compression is to be used, then the primary-key field must be at the beginning of the record.
- If the primary-key field is the trailing field in the record, the primary-key values can be of variable lengths.
- If the key field is to be treated as a data type other than STRING, the <offset> should be chosen so the field begins on on a word boundary.

INDEX BLOCKS. The length, in bytes, of a DP1 file's index blocks must be specified. The block size must be a multiple of sector size (512 bytes) and can not be greater than 4096 bytes. For DP2 files, the index block size is the same as the file's data-block size, which must be 512, 1024, 2048, or 4096 bytes.

If a DP1 file's index-block size is not specified, it defaults to the data-block size. With DP2, it always defaults to the specified data-block size, regardless of whether you have specified an index-block size.

When choosing block lengths, remember that longer index blocks require more space in the cache buffer but a longer index block may reduce the number of indexing levels and, therefore, accesses to the disc.

Considerations for Files Having Alternate Keys

Files having alternate keys involve several important considerations, including:

- Type of disc process
- Unique alternate keys
- Key specifier
- Alternate-key files
- Key length
- Key offset
- Null value
- No automatic updating

TYPE OF DISC PROCESS. A primary-key file and its associated alternate-key files must all be on the same kind of disc volumes --DP1 or DP2. They cannot be mixed.

UNIQUE ALTERNATE KEYS. An alternate-key field can be designated to require a unique value in each record. If an application tries to insert a record that duplicates an existing record's value in a unique key field, the insertion is rejected. A "duplicate record" error indication will be returned to the application program in a subsequent call to FILEINFO. With nonunique alternate keys, the insertion above would be permitted.

If a file has one or more unique alternate keys, remember that:

- For each unique alternate key having a different <key length>, the user must create a separate alternate-key file.
- More than one unique alternate key of the same <key length> can be referred to by the same alternate-key file.

KEY SPECIFIER. To identify a particular key field as an access path when positioning, each key field must be uniquely identified among other key fields in a record by a two-byte key specifier. The key specifier for primary keys is predefined as ASCII "<null><null>" (binary zero). Key specifiers for alternate-key fields are application-defined and must be supplied when the primary file is created.

ALTERNATE-KEY FILES. For each primary structured file having one or more alternate keys, you must create at least one corresponding alternate-key file.

Each record in an alternate-key file refers to only one alternate key, but the file can contain references to one or more alternate keys. Thus, with five alternate keys, the alternate-key file would have five records for each primary-file record. The primary file can have multiple alternate-key files; for example, one might contain references to three of the alternate keys, with a second alternate-key file containing references to the other two keys.

Some reasons to have separate alternate-key files are:

- Each individual alternate-key file is smaller than a combined file with several alternate-key references, so fewer index references are needed to locate a given alternate key.
- Frequent updating of one alternate key can cause fragmentation of the file. With separate files, this fragmentation would not affect references to the other keys.
- A unique alternate key cannot share a file with other keys of different lengths.

Some reasons not to have separate alternate-key files are:

- System control-block space is allocated for each opening of an alternate-key file (that is, each opening of the primary file).
- A file control block is allocated for the first opening of an alternate-key file.

Alternate-key files can be partitioned to span multiple volumes.

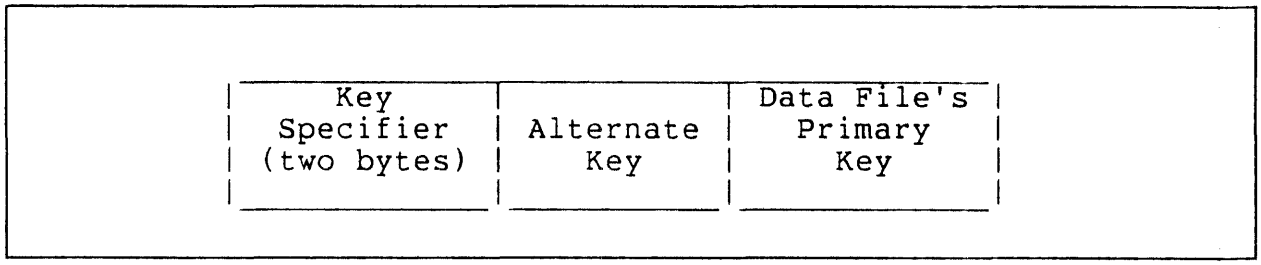


Figure 4-1. Record Structure of an Alternate-Key File

The record structure of an alternate-key file is shown in Figure 4-1. Its record length is

```
  2  for the <key specifier>
+ the <key length> of the longest alternate key included in
    the record
+ the <key length> of the associated primary key
```

An alternate-key file is itself a key-sequenced file, so the maximum record length under the DP1 disc process is limited to

$$\frac{\text{<data-block size>} - 26}{2}$$

Under the DP2 disc process, the maximum record length is

<block size> - 34

KEY LENGTH. The primary-key length of an alternate-key file--distinguished from the data file's primary key--depends on whether the file contains unique key references. With nonunique key references, the file's primary key is the entire record, so its primary-key length is the same as its record length.

If an alternate-key file contains a single, nonunique key, then, that key can be no longer than

```
255  (maximum <key length> of the alternate-key file)
- 2  for the <key specifier>
- the <key length> of the data file's primary key
```

Thus, if the data file's primary key is 33 bytes long, no nonunique alternate key can be more than 255 - 2 - 33 = 220 bytes long.

If the alternate-key file contains unique key references, its primary key is the key specifier and the unique key. Therefore, the primary-key length is

```
  2 for the <key specifier>
+ <key length> of the unique alternate-key field
```

Thus, a unique alternate key can be as long as 255 - 2 = 253 bytes, regardless of the data file's primary key.

KEY OFFSET. For each alternate key, you must specify the offset in the record--where the alternate-key field begins.

Some considerations when choosing the offset of an alternate-key field are:

- An alternate-key field can begin at any offset in the record.
- Alternate-key fields can overlap each other and the primary-key field of a key-sequenced file.
- Alternate-key fields are fixed-length but need not be written when inserting or updating a record.
- If any part of a given alternate-key field is present when inserting or updating a record, the entire field must be present.
- If the key field is to be treated as a data type other than STRING, the <offset> should be chosen so the field begins on a word boundary.

NULL VALUE. Any alternate key can be assigned a null value. When a record is inserted, if each byte in such an alternate-key field contains the null value, the alternate-key reference is not added to the alternate-key file. In the case of an update, if such an alternate-key field is changed to contain only bytes of the null value, the alternate-key reference is deleted from the alternate-key file.

If the file is read sequentially via such an alternate key, records containing the null value will not be found. Instead, the record returned (if any) is the next record in the access path not having the null value in the alternate-key field.

The most common null values are ASCII blank (%40) and binary 0.

NO AUTOMATIC UPDATING. The data-base designer can designate that the alternate-key-file contents for an alternate key not be automatically updated by the system when the value of an alternate-key field changes.

Some reasons for not having automatic updating by the system are:

- Certain fields may not be referred to until a later date. Therefore, they can be updated in a batch (one-pass) mode more efficiently.

- A field can have multiple null values. In this case, the application program must have the alternate-key file open separately. The program must determine whether the field contains a null value. If it does not, the application program then inserts the alternate-key reference into the alternate-key file.

CREATION EXAMPLES

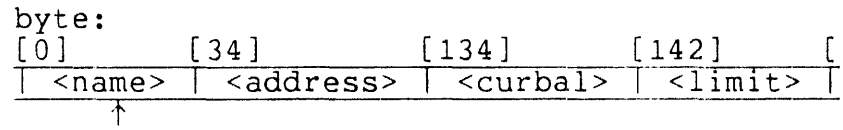
The next eight pages contain file-creation examples for five kinds of DP1 files:

1) key-sequenced file
2) key-sequenced file with alternate keys
3) alternate-key file for programmatically created primary
4) partitioned relative file
5) partitioned key-sequenced file

If these were DP2 files, IBLOCK would be ignored because all blocks in a DP2 file are the same size. They could also have additional parameters specified, such as MAXEXTENTS or BUFFERED.

Example 1: Key-Sequenced File

To create a key-sequenced file for this record:



using FUP, you could use these commands:

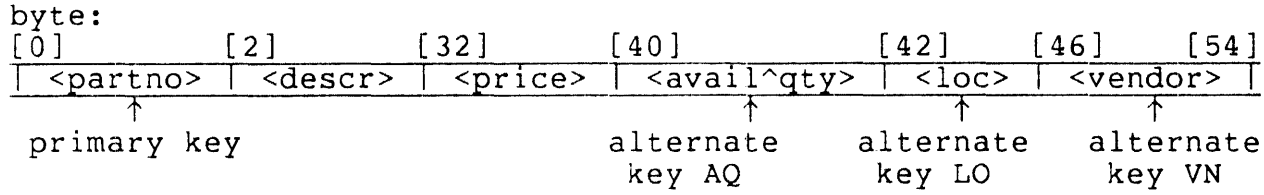
```
:FUP
GUARDIAN FILE UTILITY PROGRAM - T9000E05 - (01OCT82) SYSTEM \AB
-SET TYPE K
-SET CODE 1000
-SET EXT (16,1)
-SET REC 150
-SET BLOCK 2048
-SET COMPRESS
-SET IBLOCK 2048
-SET KEYLEN 34
-SHOW
    TYPE K
    CODE 1000
    EXT ( 16 PAGES, 1 PAGES )
    REC 150
    BLOCK  2048
    IBLOCK 2048
    KEYLEN 34
    KEYOFF 0
    DCOMPRESS, ICOMPRESS
-CREATE myfile
CREATED - $STORE1.SVOL1.MYFILE
```

Using the CREATE procedure, you could write this in an application program:

```
INT .cust^filename [0:11] := "$STORE1 SVOL1   MYFILE  ";

    .key^parameters [0:2] := [ 34,    ! key length.
                               0,     ! key offset.
                               0, ];! index-block size, uses
                                    ! the data-block size.

LITERAL
     pri^extent = 16,              ! primary-extent size = 16*2048
     file^code = 1000,
     sec^extent = 1,               ! secondary-extent size=1*2048.
     file^type = %33,              ! file type=key-sequenced, with
                                   !   data and index compression.
     rec^len = 150,                ! record length.
     data^block^len = 2048;        !
  .
  .
CALL CREATE (cust^filename, pri^extent, file^code, sec^extent,
          file^type, rec^len, data^block^len, key^parameters);
IF < THEN ... ! error.
```

Example 2: Key-Sequenced File with Alternate Keys

To create a DP1 key-sequenced file for this INVENTORY record:



using FUP, you could use these commands:

```
-SET TYPE K
-SET CODE 1001
-SET EXT (32,8)
-SET REC 54
-SET BLOCK 4096
-SET IBLOCK 1024
-SET KEYLEN 2
-SET ALTKEY ("AQ",KEYOFF 40,KEYLEN 2)
-SET ALTKEY ("LO",KEYOFF 42,KEYLEN 4)
-SET ALTKEY ("VN",KEYOFF 46,KEYLEN 8)
-SET ALTFILE (0,INVALT)
-SHOW
    TYPE K
    CODE 1001
    EXT ( 32 PAGES, 8 PAGES )
    REC 54
    BLOCK  4096
    IBLOCK 1024
    KEYLEN 2
    KEYOFF 0
    ALTKEY ( "AQ", FILE 0, KEYOFF 40, KEYLEN 2 )
    ALTKEY ( "LO", FILE 0, KEYOFF 42, KEYLEN 4 )
    ALTKEY ( "VN", FILE 0, KEYOFF 46, KEYLEN 8 )
    ALTFILE ( 0, $STORE1.SVOL1.INVALT )
    ALTCREATE
-CREATE inv
CREATED - $STORE1.SVOL1.INV
CREATED - $STORE1.SVOL1.INVALT
```

The DP2 disc process ignores the IBLOCK specification. If this were a DP2 file, all blocks in this file would be 4096 bytes long.

Using the CREATE procedure, you could write this in an application program:

```
INT .inv^filename [0:11] := "$STORE1 SVOL1   INV     ";
    .pri^key [0:2]   := [ 2,            ! key length = 2.
                          0,            ! key offset = 0.
                          1024 ];       ! index-block len.=1024.

    .alt^keys [0:24] := [ %000403,      ! 1 alternate-key file,
                                        ! 3 alternate keys.

                          ! key description for key 1.

                          "AQ",         ! key specifier = "AQ".
                          40,           ! key offset = 40.
                          2,            ! key length = 2.
                          0,            ! key file number.

                          ! key description for key 2.

                          "LO"          ! key specifier = "LO".
                          42,           ! key offset = 42.
                          4,            ! key length = 4.
                          0,            ! key file number.

                          ! key description for key 3.

                          "VN"          ! key specifier = "VN".
                          46,           ! key offset = 46.
                          8,            ! key length = 8.
                          0,            ! key file number.

                          ! key file name

                          "$STORE1 ",   ! volume,
                          "SVOL1   ",   ! subvolume,
                          "INVALT  "]; ! disc file name.

LITERAL
     pri^extent = 32,              ! primary extent size= 32*2048.
     file^code = 1001,
     sec^extent = 8,               ! secondary extent size=8*2048.
     file^type = %03,              ! file type = key-sequenced.
     rec^len = 54,                 ! record length = 54.
     data^block^len = 4096;        ! data-block size = 4096.

CALL CREATE (inv^filename, pri^extent, file^code,
             sec^extent, file^type, rec^len,
             data^block^len, pri^key, alt^keys);
IF < THEN ... ! error.
```

Note that the alternate-key file must be created separately.

Example 3: Alternate-Key File

The alternate-key file for Example 2's key-sequenced file is created automatically when FUP is used for file creation.

If the primary file is created programatically, the alternate file must be created in a separate operation. To create the alternate-key file for Example 2's key-sequenced file, you could write this sequence in an application program:

```
  INT .alt^filename [0:11] := "$STORE1 SVOL1   INVALT  ",

      .pri^key[0:2] := [ 12,      !    maximum alt.-key length
                                  ! +  primary-key length
                                  ! +  2.
                         0,       ! key offset = 0.
                         1024 ]; ! index-block size = 1024.

  LITERAL
       pri^extent = 32,           ! primary-extent size = 32*2048
       file^code = 1002,
       sec^extent = 8,            ! secondary-extent size=8*2048.
       file^type = %13,           ! file type = key-sequenced,
                                  !    with data compression.
       rec^len = 12,              ! record length = 12.
       data^block^len = 4096;     ! data-block size = 4096.

     .
     .
     .
  CALL CREATE (alt^filename, pri^extent, file^code,
               sec^extent, file^type, rec^len,
               data^block^len, pri^key);
  IF < THEN ... ! error.
     .
     .
     .
```

Example 4: Relative, Partitioned File

To create a relative file with a record length of 128 bytes that spans four partitions, using FUP, you could use these commands:

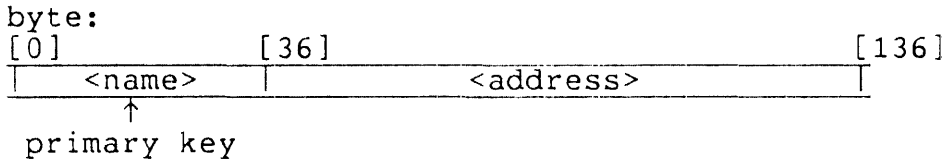
```
-SET TYPE R
-SET EXT (64,8)
-SET REC 128
-SET BLOCK 4096
-SET PART (1,$PART1,64,8)
-SET PART (2,$PART2,64,8)
-SET PART (3,$PART3,64,8)
-SHOW
    TYPE R
    EXT ( 64 PAGES, 8 PAGES )
    REC 128
    BLOCK  4096
    PART ( 1, $PART1, 64, 8 )
    PART ( 2, $PART2, 64, 8 )
    PART ( 3, $PART3, 64, 8 )
-CREATE filea
CREATED - $PART0.USERA.FILEA
```

Using the CREATE procedure, you could write this in an application program:

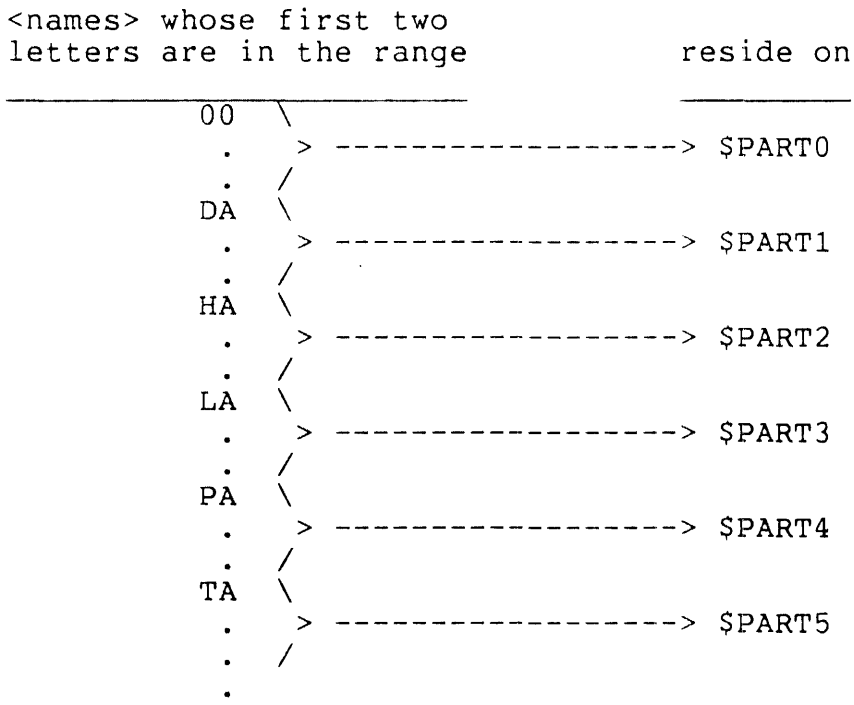
```
INT .rel^filename[0:11] := "$PART0  USERA   FILEA   ",
                                  ! partition parameters array
    .partarray [0:17] := [ 3,     ! number of extra partitions
                      "$PART1  ", ! volume name of first extra
                      "$PART2  ", ! volume name of second extra
                      "$PART3  ", ! volume name of third extra
                      64,         ! primary ext. for first extra
                      64,         ! pri. extent for second extra
                      64,         ! pri. extent for third extra
                      8,          ! second. ext. for first extra
                      8,          ! sec. extent for second extra
                      8 ];        ! sec. extent for third extra

LITERAL
     pri^extent = 64,           ! primary extent size = 64*2048
     sec^extent = 8,            ! secondary extent size=8*2048
     file^type = %01,           ! file type = relative.
     rec^len = 128,             ! record length = 128.
     data^block^len = 4096;     ! data-block size = 4096.
  .
  .
  .
CALL CREATE (rel^filename, pri^extent,,
             sec^extent, file^type, rec^len,
             data^block^len,,,partarray);
IF < THEN ... ! error.
```

Example 5: Key-Sequenced, Partitioned File

This example partitions a key-sequenced file having this record format:



The file resides on six volumes and is partitioned in this manner:



Using FUP to create this file, you could use these commands to describe the partitioning:

```
-SET PART (1,$PART1,64,8,"DA")
-SET PART (2,$PART2,64,8,"HA")
-SET PART (3,$PART3,64,8,"LA")
-SET PART (4,$PART4,64,8,"PA")
-SET PART (5,$PART5,64,8,"TA")
```

Using the CREATE procedure to create this file, the partitioning would be described in this partition array:

```
                                ! partition-parameters array
 .partarray [0:36] := [ 5,      ! number of extra partitions
                    "$PART1  ", ! volume name of first extra
                    "$PART2  ", ! volume name of second extra
                    "$PART3  ", ! volume name of third extra
                    "$PART4  ", ! volume name of fourth extra
                    "$PART5  ", ! volume name of fifth extra
                    64,         ! primary extent for first extra
                    64,         ! primary extent for second extra
                    64,         ! primary extent for third extra
                    64,         ! primary extent for fourth extra
                    64,         ! primary extent for fifth extra
                    8,          ! secondary extent for first extra
                    8,          ! secondary extent for second extra
                    8,          ! secondary extent for third extra
                    8,          ! secondary extent for fourth extra
                    8,          ! secondary extent for fifth extra
                    2,          ! partial-key length = 2
                    "DA",       ! key value for $PART1
                    "HA",       ! key value for $PART2
                    "LA",       ! key value for $PART3
                    "PA",       ! key value for $PART4
                    "TA"        ! key value for $PART5
                    ];
```

SECTION 5

FILE ACCESS

Topics discussed in this section are:

- Opening files
- Access rules for structured files
- Access rules for unstructured files
- Other considerations for both structured and unstructured files, such as locking records and files, verifying write operations, refreshing, and programmatic allocation and deallocation of extents
- Error-recovery considerations for key-sequenced files, for files having alternate keys, and for partitioned files
- Access examples

File creation is discussed in Section 4, "File Creation," and in the descriptions of the File Utility Program (FUP) in the GUARDIAN Operating System Utilities Reference Manual and GUARDIAN Operating System User's Guide.

OPENING AND CLOSING A FILE

Communication is established with a disc file through use of the OPEN procedure. For example, this code opens a permanent disc file:

```
  ...
INT .file^name[0:11] := "$VOL2   STORE1  TRANFILE";

CALL OPEN ( file^name, file^number,,1 ); ! wait I/O,
                                         ! shared access,
                                         ! read/write access,
                                         ! sync depth = 1.
IF <> THEN ...
```

Subsequently, the file is identified to other procedures by the file number returned by the OPEN procedure.

Access to a disc file is terminated by use of the CLOSE procedure, as shown here:

```
  ...
CALL CLOSE ( file^number );
  ...
```

Communication is established with a temporary file by passing the array containing the temporary file name, returned from CREATE, to the OPEN procedure:

```
CALL OPEN ( temp^file, temp^file^number,, 1 ); ! wait I/O,
                                               ! shared access,
                                               ! read/write access,
                                               ! sync depth = 1.
IF <> THEN ...
```

As with a permanent file, other procedures access the temporary file by using the file number returned by OPEN.

A temporary file is purged when you close it. In the example above,

```
  ...
CALL CLOSE ( temp^file^number );
  ...
```

would delete the temporary file from volume $VOL2.

If you do not want the file purged when it is closed, a temporary file can be made permanent by use of the RENAME procedure. In this example,

```
  ...
new^name ':=' "$VOL2   STORE1  NEWFILE ";
CALL RENAME ( temp^file^number, new^name );
IF < THEN ....
  ...
CALL CLOSE ( temp^file^number );
  ...
```

the temporary file is renamed, making it permanent. The volume name supplied to RENAME must be the same as that used when the temporary file was created, but both the subvolume and the file name can be different.

OPENING PARTITIONED OR ALTERNATE-KEY FILES

When the first partition of a partitioned file is opened, all partitions are automatically opened. If one of the partitions cannot be opened, access to the file is still granted but OPEN returns a CCG warning indication (see "Condition Codes" in Section 3) and some operations may not work. You can call the FILEINFO procedure to obtain the file-system error number and the FILERECINFO procedure to identify the highest-numbered partition that did not open.

Individual partitions cannot be opened separately unless "unstructured access" (OPEN <flags>.<2> = 1) is specified. See the OPEN procedure in the System Procedure Calls Reference Manual for details on unstructured access.

If the file has one or more alternate keys, all alternate-key files are automatically opened when the primary file is opened. If an alternate-key file cannot be opened, OPEN returns a CCG warning indication. You can then call the FILEINFO procedure to obtain the file-system error number and the FILERECINFO procedure determine which key file couldn't be opened. The file is still accessible. However, an attempt to use an access path associated with an alternate-key file that did not open results in an error 46 ("invalid key specified").

Alternate-key files can be opened and accessed separately from their primary files.

If the file is not partitioned and does not have alternate keys, there are no special considerations.

ACCESS TYPES (DP2 Files Only)

Buffer and cache management under the DP2 disc process are more efficient if you specify the proper access type when you open a file. Which one you should specify depends on how you intend to use the file.

- If you specify system-managed access, the default access type, the disc process uses previous I/O requests as a basis for predicting whether the user is performing random or sequential access and optimizes its behavior accordingly.

- If you specify random access, the disc process employs a "least recently used" (LRU) method when reusing cache space. This helps ensure that a frequently used block can remain in the cache instead of being read in each time it is needed.

- If you specify sequential access, the disc process allocates a single cache buffer, reusing it when the next data block is requested. This prevents the cache space from being filled with data blocks that the user probably will not need again.

- If you specify direct I/O, the disc process bypasses the cache completely when the file has been opened with write-through, unstructured access in either exclusive read/write or protected read-only mode. Direct I/O could be desirable in an application which requires fast unstructured access and receives no benefit from the cache mechanism.

The SETMODE procedure lets you set or examine the access type for any DP2 file.

END-OF-FILE POINTER

An end-of-file (EOF) pointer is associated with each disc file. This pointer contains the relative byte address of the first byte of the next available block. When appending to a file, the end-of-file pointer is advanced automatically each time a new block is allocated at the end of the file.

The system maintains the working copy of a file's end-of-file pointer in the file control blocks (FCBs) that are in both of the two system processes that control the associated disc volume. A file's end-of-file address is physically written on disc every time one of these events occurs:

- The file is created.
- An extent is allocated for the file.
- A CONTROL operation is performed for the file.
- The last accessor closes the file.
- The REFRESH procedure is called for the file.
- The PUP command REFRESH is executed for the file's volume.
- The end-of-file pointer is changed (if the autorefresh option is in effect).
- A block split or collapse occurs in a DP2 key-sequenced file.

The autorefresh option can be specified when the file is first created. Even if a file is created without the autorefresh option specified, it can be modified at some later date to include the autorefresh option. DP2 key-sequenced files always have this option on.

When creating a file with the CREATE procedure, you can specify the autorefresh option by setting <file type>.<12> = 1. When creating a file with FUP, you can specify the autorefresh option with the SET REFRESH command. For files created without the autorefresh option, the option can specified at any time with the FUP command ALTER ... REFRESH ON.

The "Refreshing" subsection below discusses when and how frequently files should be refreshed.

AUDIT-CHECKPOINT COMPRESSION (DP2 Files Only)

If a file is audited or is opened with "sync depth" greater than 0, updating a record in that file causes an audit-checkpoint (AC) record to be created. The AC record, describing changes incurred for the update, is sent to the backup disc process. If the file is audited by TMF, the AC record also is sent to the audit trail's disc process.

If the file is audited, the AC record gets copied from place to place several times. It occupies permanent space in the audit-trail file and also occupies resident memory in the backup disc process until the backup determines that the updated record has been written to disc. If the file is not audited, the AC record is sent as a checkpoint message to the backup disc process, where it occupies resident memory until it is no longer needed for takeover recovery.

An AC record includes (1) an AC-record header approximately 64 bytes long, (2) a copy of the record before the update, or before-image, and (3) a copy of the record after the update, or after-image. If a data record is 1000 bytes long, therefore, the AC record for its update would be 2064 bytes long.

When you create a DP2 file, you can specify audit-checkpoint compression to lessen this overhead. Later, you can turn this feature on or off by using the SETMODE procedure or the File Utility Program (FUP) command ALTER.

AC compression shortens parts (2) and (3) of the AC record by omitting the changed fragments of the data record. The AC record must include the record key (in a key-sequenced file), record number (in a relative file), or relative byte address (in an entry-sequenced or unstructured file), but no other unchanged

fields. For example, if only two 10-byte fields in a 1000-byte record were updated, the compressed AC record could be about 130 bytes long instead of 2064 bytes. Although AC compression uses some extra CPU cycles, it has several advantages:

- CPU and memory cycles are reduced during message transferral, although some extra cycles are required to perform the compression. If the file in our example is audited, the compressed AC record causes only 1040 bytes of I/O (8 * 130) instead of 16,512 bytes (8 * 2064).

- Resident memory requirements in the backup CPU are reduced.

- If the file is audited, audit-trail consumption is reduced and audit-blocking is more efficient because of the smaller AC records.

When you create a data file of any type, you can establish whether AC compression should be enabled or disabled when the file is opened. After you open the file, you can use the SETMODE procedure to enable or disable AC compression.

Even when AC compression is enabled for a file, not every AC record is compressed. Also, some limits are imposed to keep the space used for recording the compression from becoming greater than the unchanged fragments that would be omitted.

- If the record length is less than a certain limit, the disc process does not compress the record. This limit, 20 bytes in the B00 release, is subject to change.

- If an update changes the record length, the disc process does not compress the AC record.

- If two changed fragments of a record are sufficiently close to each other, they and the bytes between them are considered one changed fragment.

- If more than a certain limit of fragments are changed, the entire remainder of the record is considered the last fragment.

- After the changed fragments are identified, the total size of the prospective compressed AC record is computed and compared to the size of a noncompressed AC record. If the savings are insufficient, the noncompressed AC record is used.

Key-sequenced records with large keys reduce the effectiveness of AC compression because the key must be kept in the AC record.

ACCESS RULES FOR STRUCTURED FILES

The best way to get access to a structured file depends on what you want to do with that file.

Sequential Access

Sequential processing, in which a related group (subset) of records is to be read in ascending order using the current access path, is accomplished by the READ and READLOCK procedures.

The records comprising a subset are indicated by the file's current positioning mode: approximate, generic, or exact. A subset can be all or part of a file or it can be empty. An attempt to read beyond the last record in a subset or to read an empty subset returns an end-of-file indication.

The first call to READ or READLOCK, after a file-opening or positioning operation, reads the record (if any) at the current position. Subsequent calls to READ or READLOCK, without intermediate positioning, return successive records (if any) in the designated subset.

Sequential reading of a relative file, after a call to OPEN, POSITION, or approximate KEYPOSITION by primary key, reads the file sequentially and skips omitted or deleted records.

After each call to READ or READLOCK, the position of the returned record becomes the current position.

Random Access

The update procedures, READUPDATE, WRITEUPDATE, READUPDATELOCK, and WRITEUPDATEUNLOCK, are used for random-access processing. The updating operation occurs at the record indicated by the current position. Random processing implies that a record to be updated must exist. Therefore, if no record exists at the current position (as indicated by an exact match of the current key value with a value in the key field designated by the current-key specifier), a "record not found" error (<error> = 11) is returned.

WRITEUPDATE or WRITEUPDATEUNLOCK cannot be used to alter a record's primary key. If this is to be done, the record must first be deleted, then inserted (via WRITE) using the new value of the primary key.

If updating or locking is attempted immediately after a call to KEYPOSITION where a non-unique alternate key is specified, the updating or locking fails with an "invalid key" error (<error> = 46). However, if an intermediate call to READ or READLOCK is performed, the updating or locking is permitted.

Inserting Records

Record insertion is accomplished via the WRITE procedure. Insertion implies that no other record exists with the same primary-key value as the record being inserted. Therefore, if such a record already exists, the operation is not performed and a "duplicate record error" (<error> = 10) is returned.

If an alternate key has been declared to be unique and an attempt is made to insert a record having a duplicate value in such an alternate-key field, the operation is not performed and a "duplicate record error" (<error> = 10) is returned.

Insertion of an empty record (one where <write count> = 0) is not valid for key-sequenced and relative files, but is valid for entry-sequenced files. The length of a record to be inserted must be less than or equal to the record length defined for the file. If not, the insertion is not performed and an "invalid count" error (<error> = 21) is returned.

Deleting Records

Record deletion (that is, WRITEUPDATE or WRITEUPDATEUNLOCK where <count> = 0) always applies to the current position in a file.

Alternate Keys

Alternate-key fields are fixed-length but need not be written when inserting or updating a record. If any part of a given alternate-key field is present when inserting or updating a record, the entire field must be present.

Current Position

The current position is subject to change only after a call to

- READ or READLOCK from any structured file
- WRITE for a relative or entry-sequenced file
- KEYPOSITION for a key-sequenced, relative, or entry-sequenced file
- POSITION for a relative, entry-sequenced, or unstructured file

After a call to READ or READLOCK, the current position becomes the position of the record just read. After a call to WRITE for a relative or entry-sequenced file, the current position becomes the position of the record just written.

Current Key Value

Except for insertions to key-sequenced files, the current key value will be set to the value of the record transferred.

Current Primary-Key Value

A file's current primary-key value is taken from the primary key associated with the last

- READ or READLOCK operation from any structured file
- WRITE operation to a relative or entry-sequenced file
- KEYPOSITION operation by primary key for a key-sequenced file
- POSITION operation by primary key for a relative or entry-sequenced file

Sequential Block-Buffering

In a NonStop 1+ system, a process can optionally specify (when a file is opened) that the file system should use an array in the process's data area for deblocking records. In a NonStop system, the array is in the process file segment (PFS), not in the data stack. The use of such an array is called sequential block-buffering.

Without sequential block-buffering, the system separately requests each record from a disc process; for each record, this involves sending an interprocess message, changing the environment, and possibly waiting to obtain some of the system data space. With sequential block-buffering, an entire block is returned from the disc process and stored in the application process's data area or in the PFS. Once a block is there, subsequent access to records within that block is performed by file-system code, not the disc process, and requires no disc access, no communication with the disc process, and no environment changes. Therefore, sequential reading from a block buffer can eliminate many messages.

Reading the buffered data does not use the disc process until:

- The block has been traversed, at which time the disc process fetches another block.
- An intervening POSITION or KEYPOSITION is performed. In this case, the next READ request causes a new block to be fetched.
- A disallowed request (such as READUPDATELOCK, READLOCK, or READUPDATE) is specified. In this case, the single record requested is read from the disc file to the user process, as in a normal procedure call, and the buffer is cleared.

CAVEATS. Note that this option is meaningful only for sequential reading of multiple records of a structured file. Neither random reading nor any writing can take advantage of the sequential buffer. In fact, writing operations automatically clear the buffer, because they must go to the disc process.

Also, you should be very careful when using the buffer with a key-sequenced file, because record access within the buffer is unstructured.

Sequential block-buffering ignores any record locks that may be in effect for the block. It does not, however, bypass file locks.

To change a record that has been read from a block buffer, you should first perform intervening POSITION, KEYPOSITION, READLOCK, or READUPDATELOCK operations to fetch the record via normal methods. This ensures that the record has not been altered or deleted by another user since the block was read. It also ensures that the record is not currently locked by another process and locks out other processes from the record to be updated.

OPEN PARAMETERS. NonStop systems and NonStop 1+ systems differ in their handling of these buffers. In either system, however, the OPEN procedure's <sequential block buffer> and <buffer length> parameters govern creation of a buffer.

In a NonStop 1+ system, the <sequential block buffer> parameter contains the address of a storage area to be used as the block buffer. The <buffer length> parameter must be greater than or equal to the data-block length of the primary file or any associated alternate-key files. (The data-block length is specified when a file is created.) If the specified length is too short, the OPEN operation succeeds but returns a CCG indication (a subsequent call to FILEINFO returns <error> = 5, a "failure to provide sequential buffering" warning) and the application process's sequential buffer is not used; instead, normal system buffering is used. For example:

```
INT .seq^buffer [ 0:2047 ];  ! sequential block buffer.
  ...
flags := %2060; ! read-only, protected, wait I/O.
CALL OPEN ( file^name,file^number,flags,,,,seq^buffer,4096 );
IF < THEN   ! OPEN failed.
  ...
ELSE
IF > THEN
  BEGIN   ! file successfully opened
    CALL FILEINFO ( file^number , error );
    IF error = 5 THEN   ! sequential buffer request rejected.
      ...
  END   ! the file is successfully opened
```

The file is then read sequentially in the normal manner.

```
eof := 0;
WHILE NOT eof DO
  BEGIN
    CALL READ ( file^number, buffer, reclen, countread );
    IF > THEN eof := 1
    ELSE
      ...
  END;
```

When READ is called and the sequential block buffer is empty (either this is the first record read from the file or all records in the block have been read), the file system transfers a data block from the disc-I/O process to the sequential block buffer ("seq^buffer") in the application process's data area. For each call to READ, a record is deblocked from the sequential block buffer and transferred into the array called "buffer".

In a NonStop system, however, the <sequential block buffer> parameter serves only as a numeric buffer identifier, because the

file system allocates the buffer space from the PFS. This parameter can be omitted if buffer space is not to be shared (see "Sharing Buffer Space" below); <buffer length> is the more significant parameter because:

- If the <buffer length> parameter is zero or absent, or is longer than the space available in the PFS, the OPEN operation succeeds but returns a CCG indication with error 5 (a "failure to provide sequential buffering" warning) and block-buffering is not used.

- If the <buffer length> parameter is greater than the file's block size, the buffer will be created with the specified size.

- If the <buffer length> parameter is nonzero but not greater than the file's block size, the buffer size will equal the block size. For example, if a file with block size 4096 is opened with <buffer length> 128 specified, the buffer will be created for a block size of 4096. This is often the easiest way to provide the most efficient usage of buffer space.

ALTERNATE-KEY FILES. If you want to use an alternate-key access path and the alternate-key file's block size is larger than that of the primary file, be certain to open the primary file with the larger <buffer length> parameter.

If access to a primary file uses sequential block-buffering, so does access to its alternate-key records.

After KEYPOSITION with a nonzero key specifier, the first READ request causes the disc process to fetch a data block from the alternate-key file into the buffer area. The disc process then fetches a single record from the primary data file via the alternate-key specification in the buffer. Thus the benefits and limitations of sequential block-buffering apply to the alternate-key-file I/O, not to the primary-file I/O.

SHARED FILE ACCESS. For sequential block-buffering, the file usually should be opened with protected or exclusive access. Combining sequential block-buffering and shared access is allowed on either a NonStop system or a NonStop 1+ system, but you should beware that this combination can cause some concurrency problems.

If another process is updating data copied into the block buffer, those updates may not be seen by the process using the buffer. For example, assume process A is reading a buffered block of data while process B inserts a new record into that block on the disc. The new record will not be in the buffer that process A is reading. Although process A's user might expect to see the

record that process B inserted, that record will not be in the buffer unless process A reads that block again.

SHARING BUFFER SPACE. In either kind of system, you can have two or more files share the same buffer space by specifying identical <sequential block buffer> parameters in OPEN. This can cause significant memory-consumption savings in some applications.

When using this feature, however, be certain that the first file opened either has the largest block size or is opened with enough buffer space to accommodate the largest file. If a file tries to share a buffer that was already created with a smaller block size, the OPEN operation succeeds but returns a CCG indication with error 5 (a "failure to provide sequential buffering" warning) and block-buffering is not used.

A shared buffer can be useful when reading whole blocks of data from several files, but it would be inefficient when reading a single record or switching back and forth between files, because the buffer is refilled each time a new file or random record is read.

If you omit the <sequential block buffer> parameter when opening a file on a NonStop system, the file cannot share a buffer.

ACCESS RULES FOR UNSTRUCTURED FILES

The best way to get access to an unstructured file depends on what you want to do with that file.

File Pointers and Relative Byte-Addressing

Data access in an unstructured disc file is via a relative byte address (RBA). The first byte in a file is at RBA zero. A relative byte address is an offset, in bytes, from the first byte.

Three pointers are associated with each open, unstructured disc file:

- The next-record pointer contains the RBA of the location where the next disc transfer (READ or WRITE) begins.
- The current-record pointer contains the RBA of the location just read or written, which is the address where a disc transfer due to a READUPDATE or WRITEUPDATE begins.

- The end-of-file pointer contains the RBA of the next even-numbered byte after the last significant data byte in a file (unless the file has ODDUNSTR set). The end-of-file pointer is incremented automatically when a record is appended to the end of a file (WRITE). It can be set explicitly by calls to the POSITION and CONTROL procedures.

Separate next-record and current-record pointers are associated with each opening of an unstructured disc file so, if the same file is open several times simultaneously, each opening provides a logically separate access. The next-record and current-record pointers reside in the file's access control block (ACB) in the application process environment.

A single end-of-file pointer, however, is associated with all openings of a given unstructured disc file. This permits data to be appended to the end of a file by several different accessors. The end-of-file pointer resides in the file's file control block (FCB) in the disc-I/O-process environment. A file's end-of-file pointer value is copied from the file label on disc when the file is opened and is not already open; the end-of-file pointer value in the file label is updated (1) when any CONTROL operation to the file is performed, (2) when a file extent is allocated for the file, and (3) when the file is closed and there are no other openings of the file.

An unstructured file's end-of-file address is physically written on disc every time one of these events occurs:

- The file is created.
- An extent is allocated for the file.
- A CONTROL operation is performed for the file.
- The last accessor closes the file.
- The REFRESH procedure is called for the file.
- The PUP REFRESH command is executed for the file's volume.
- The end-of-file pointer is changed (if the autorefresh option is in effect).

The autorefresh option can be specified when the file is first created. Even if a file is created without the autorefresh option specified, it can be modified at some later date to include the autorefresh option.

When creating a file with the CREATE procedure, you can specify the autorefresh option setting <file type>.<12> to 1. When creating a file with FUP, you can specify the autorefresh option with the SET REFRESH command. For files created without the autorefresh option, you can specify the option at any time with the FUP command ALTER REFRESH.

Table 5-1 summarizes pointer action with unstructured disc files.

Table 5-1. File-Pointer Action

In this table, <count> is the specified transfer count. If the file was created with the CREATE procedure's ODDUNSTR parameter set, the value specified for <count> is the number of bytes transferred. If the ODDUNSTR parameter was not set when the file was created, <count> is rounded up to an even number before the data are transferred.

```
CREATE

   file label's end-of-file pointer := 0D;

OPEN (first)

   end-of-file pointer := file label's end-of-file pointer;

OPEN (any)

   current-record pointer := next-record pointer := 0D;

READ

   current-record pointer := next-record pointer;
   next-record pointer := next-record pointer +
      $min (<count>, eof pointer - next-record pointer);

WRITE

   if next-record pointer = -1D then
     begin
       current-record pointer := end-of-file pointer;
       end-of-file pointer := end-of-file pointer + <count>;
     end
   else
     begin
       current-record pointer := next-record pointer;
       next-record pointer := next-record pointer + <count>;
       end-of-file pointer := $max( end-of-file pointer,
                                    next-record pointer );
     end;

READUPDATE

   file pointers are unchanged
```

Table 5-1. File-Pointer Action (continued)

```
WRITEUPDATE

   file pointers are unchanged

CONTROL (write end-of-file)

   end-of-file pointer := next-record pointer;
   file label's end-of-file pointer := end-of-file pointer;

CONTROL (purge data)

   current-record pointer := next-record pointer :=
      end-of-file pointer := 0D;
   file label's end-of-file pointer := end-of-file pointer;

CONTROL (allocate/deallocate extents)

   file pointers are unchanged
   file label's end-of-file pointer := end-of-file pointer;

POSITION

   current-record pointer := next-record pointer :=
     <relative byte address>;

CLOSE (last)

   file label's end-of-file pointer := end-of-file pointer;
```

Sequential Access

READ and WRITE increment the next-record pointer by the number of bytes transferred; therefore, automatic sequential access is provided to the unstructured file.

If the file was created with the ODDUNSTR parameter set, the number of bytes transferred and the amount the pointers are incremented are exactly the number of bytes specified with <write count> or <read count>. If the ODDUNSTR parameter was not set when the file was created, the values of <write count> and <read count> are rounded up to an even number before the transfer takes place and the file pointers are incremented by the rounded-up value.

EXAMPLE. This sequence of ENSCRIBE calls shows how the file pointers are used when sequentially accessing an unstructured disc file. Assume these are the first operations to the file after OPEN:

```
  ...
CALL READ ( file^a, buffer, 512 );
  ...
CALL READ ( file^a, buffer, 512 );
  ...
CALL WRITEUPDATE ( file^a, buffer, 512 );
  ...
CALL READ ( file^a, buffer, 512 );
  ...
```

The first READ transfers 512 bytes into "buffer" starting at relative byte 0. The next-record pointer now points to relative byte 512 and the current-record pointer points to relative byte 0.

The second READ transfers 512 bytes into "buffer" starting at relative byte 512. The next-record pointer now points to relative byte 1024 and the current-record pointer points to relative byte 512.

The WRITEUPDATE procedure then replaces the just-read data with new data in the same disc location. The file system transfers 512 bytes from "buffer" to the file, at the position indicated by the current-record pointer (relative byte 512). The next-record and current-record pointers are not affected by the WRITEUPDATE procedure.

The third READ transfers 512 bytes into "buffer" starting at relative byte 1024 (the address in the next-record pointer). The next-record pointer then points to relative byte 1536, and the current-record pointer points to relative byte 1024.

ENCOUNTERING THE END OF THE FILE DURING SEQUENTIAL READING. When the end-of-file (EOF) boundary is encountered during reading of an unstructured disc file, data up to the EOF location are transferred. A subsequent READ will return an EOF indication (condition code of CCG) because it is not permissible to read data past the EOF location. If the file is not repositioned, the EOF indication will be returned with every subsequent READ.

For example, assume an unstructured file is written on disc, with the EOF location at relative byte 4096. Sequential readings of 512 bytes each are executed, starting at relative location 0:

```
file^eof := 0;
WHILE NOT file^eof DO
  BEGIN
    CALL READ ( file^a, buffer, 512, num^read, .. );
    IF > THEN file^eof := 1
    ELSE
    IF = THEN
      BEGIN
        ...process the data...
      END
    ELSE ... ! error.
  END;
```

Each of the first eight READ calls transfers 512 bytes into "buffer", returns "num^read" = 512, and sets the condition-code indicator to CCE (operation successful).

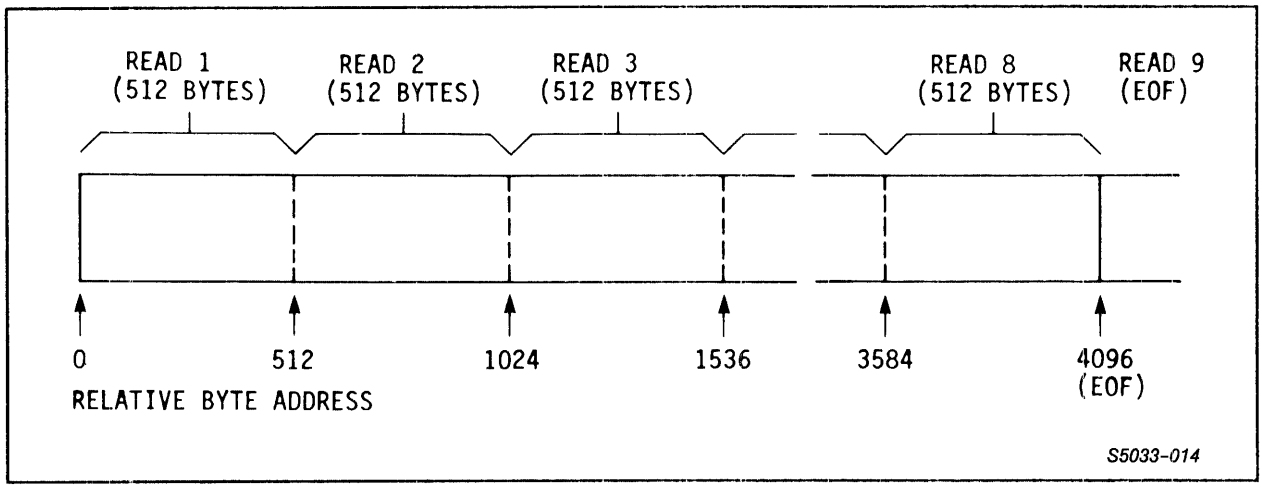


Figure 5-1. Example of Encountering EOF

The next READ fails, so no data are transferred into "buffer", "num^read" is returned as zero (0), and the condition-code indicator is set to CCG (end-of-file indication).

Figure 5-1 shows a sample EOF encounter.

If sequential READs of 400 bytes are executed from the same file, the results are slightly different:

```
file^eof := 0;
WHILE NOT file^eof DO
  BEGIN
    CALL READ ( file^a, buffer, 400, num^read, .. );
    IF > THEN file^eof := 1
    ELSE
    IF = THEN
      BEGIN
        ...process the data...
      END
    ELSE ... ! error.
  END;
```

In this case, the first 10 READ calls each transfer 400 bytes into "buffer", return "num^read" = 400, and set the condition-code indicator to CCE (operation successful). The eleventh READ transfers 96 bytes into "buffer", returns "num^read" = 96, and sets the condition-code indicator to CCE. The next READ fails and sets the condition-code indicator to CCG. This situation is illustrated in Figure 5-2.

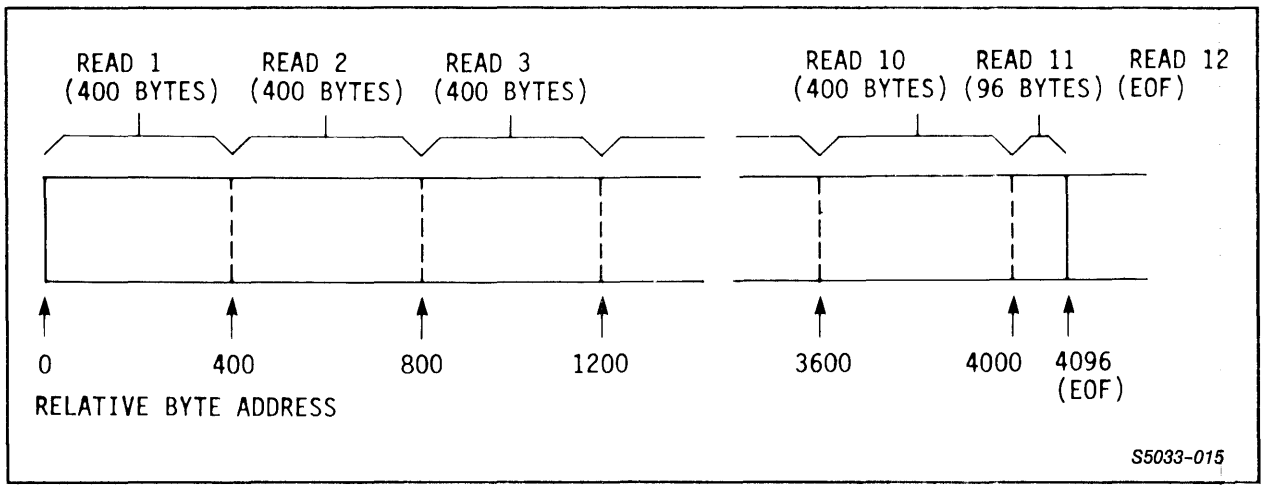


Figure 5-2. Example of Encountering EOF (Short READ)

Random Access

Random access to an unstructured disc file is accomplished by setting the file pointers explicitly. This is done by calling the POSITION procedure and specifing the starting location to be accessed in the <relative byte address> parameter.

For example, to update data in an unstructured disc file at relative byte address 81920, you could use this sequence of calls:

```
CALL POSITION ( file^a, 81920D );
  ...
CALL READUPDATE ( file^a, buffer, 512 );
  ...
CALL WRITEUPDATE ( file^a, buffer, 512 );
```

The call to POSITION sets the next-record and current-record pointers to relative byte 81920.

The call to READUPDATE transfers 512 bytes from the file to "buffer", starting at relative byte 81920. Following the read, the next-record and current-record pointers are unchanged.

The WRITEUPDATE procedure replaces the just-read data with new data in the same location on disc. The file system transfers 512 bytes from "buffer" to the file at relative byte 81920.

Appending to the End of a File

The POSITION procedure can be used to append data to the end of an unstructured disc file. To set the pointer to the current end-of-file, pass -1D as the <relative byte address> parameter:

```
CALL POSITION ( file^a, -1D );
```

The next-record pointer now contains -1D. This indicates to the file system that subsequent WRITE calls should append to the end of the file.

A subsequent WRITE, then, appends 512 bytes to the end of the file:

```
CALL WRITE ( file^a, buffer, 512, num^written );
```

The file system transfers 512 bytes from "buffer" to the current end-of-file location (131072). The next-record and end-of-file pointers now point to relative byte 131584; the current-record pointer points to relative byte 131072; the next-record pointer still contains -1D, so a subsequent WRITE also appends to the end of the file.

Figure 5-3 shows an example of using POSITION for both random access and appending to the end of an unstructured disc file.

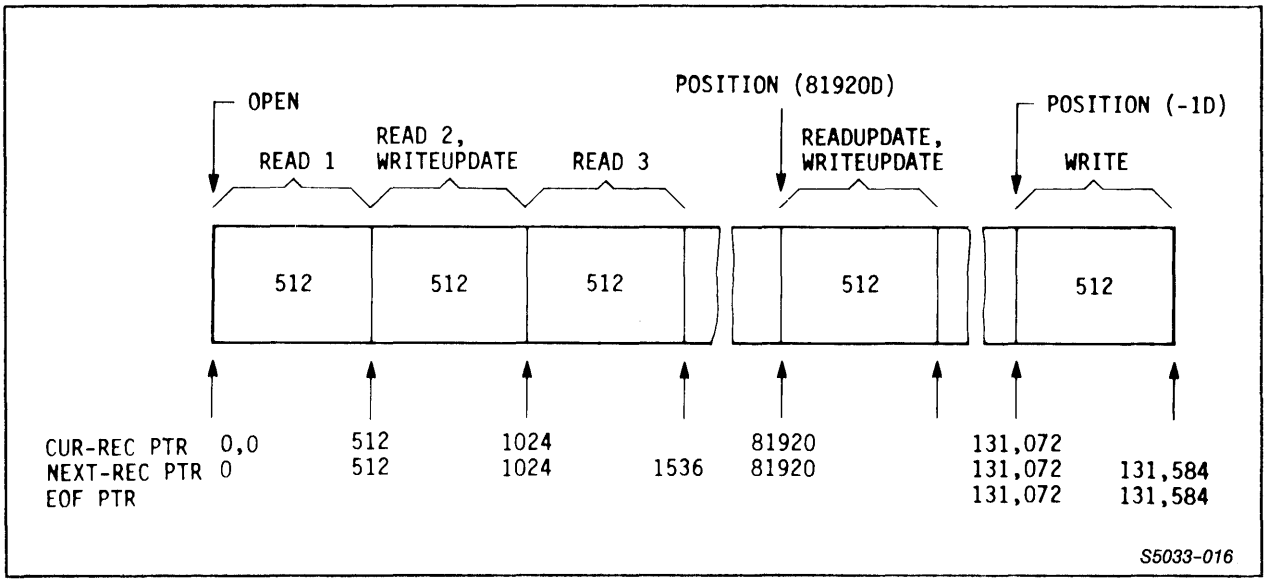


Figure 5-3. Example of File-Pointer Action

Heeding Sector or Buffer Boundaries

Disc storage is divided into physical sectors, each containing 512 bytes. Any disc READ or WRITE operation, regardless of the number of bytes transferred, involves at least one sector and can involve as many as eight sectors (4096 bytes).

It is most efficient to transfer whole sectors of information. This means the most efficient transfers are multiples of 512 bytes on 512-byte boundaries. These boundaries are easy to find because a file always starts on a sector boundary.

The DP2 disc process treats the buffer size of an unstructured file as a substitute for sector size. This buffer size must be 512, 1024, 2048, or 4096 bytes. For a DP2 unstructured file, then, the most efficient transfer starts on a <buffer-size> boundary and is an integral multiple of the buffer size. Thus, if the buffer size is 4096, you can plan efficient transfers as you would with a DP1 disc having 4096-byte sectors.

Consider a disc operation transferring two full sectors of information (1024 bytes) to an unstructured disc file, starting at relative address 512:

```
POSITION (out^file, 512);
CALL WRITE ( out^file, rodney, 1024, ... );
```

This call involves only one disc operation: the 512 bytes of "rodney" are transferred to one disc sector.

Now consider a WRITE that does not fall on sector boundaries. Suppose you write only 200 bytes, starting at relative location 400, using these calls:

```
POSITION (out^file, 400);
CALL WRITE ( out^file, king, 200 );
```

This situation, illustrated in Figure 5-4, transfers two disc sectors, even though "king" is shorter than "rodney." Also, because only part of each sector is to receive data from "king," two disc operations are required:

1. The two sectors, containing relative addresses [0:1023], are read from the disc into the disc's buffer area in main memory. The first 200 bytes of "king" are moved into the appropriate location in the disc's buffer (address [400]).

2. The updated sectors are written back to the disc.

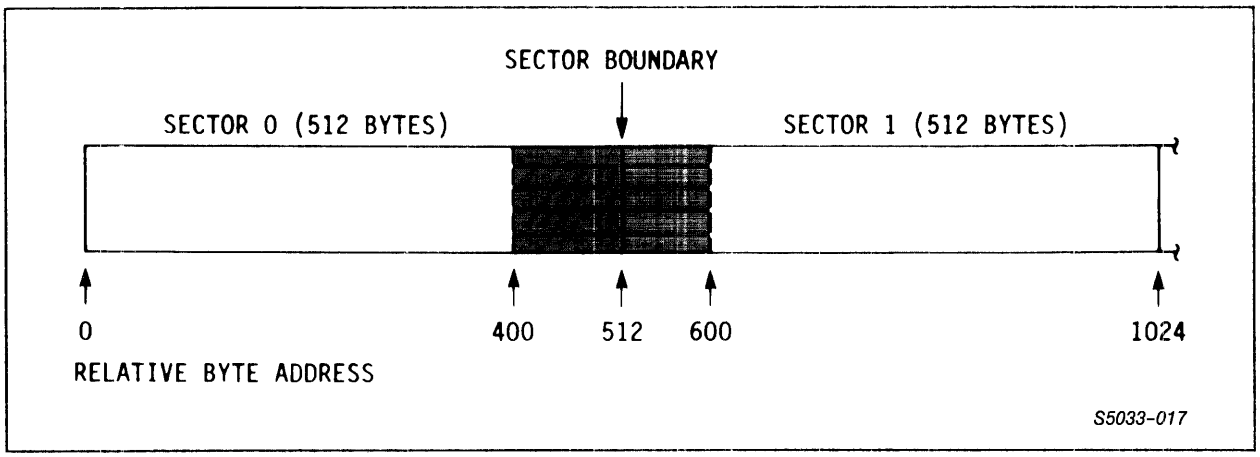


Figure 5-4. Example of Crossing Sector Boundaries

Although full-sector transfers are most efficient for the system to perform, they are not necessarily the most efficient for a particular application. If the application can block data to fill a buffer or sector, however, its I/O will be more efficient.

Resident Buffering (NonStop 1+ System Only)

For unstructured files on a Nonstop 1+ system, resident buffering can save some data transferral and avoid suspension of an application process waiting for file-system buffer space for I/O.

With resident buffering, any data transferred because of an I/O

request are transferred directly between the application process's data area and an I/O buffer in the processor running the primary I/O process controlling a device. This bypasses the normal intermediate transfer to a file-system buffer in the processor running the application process.

The effect of using resident buffering is shown in Figure 5-5.

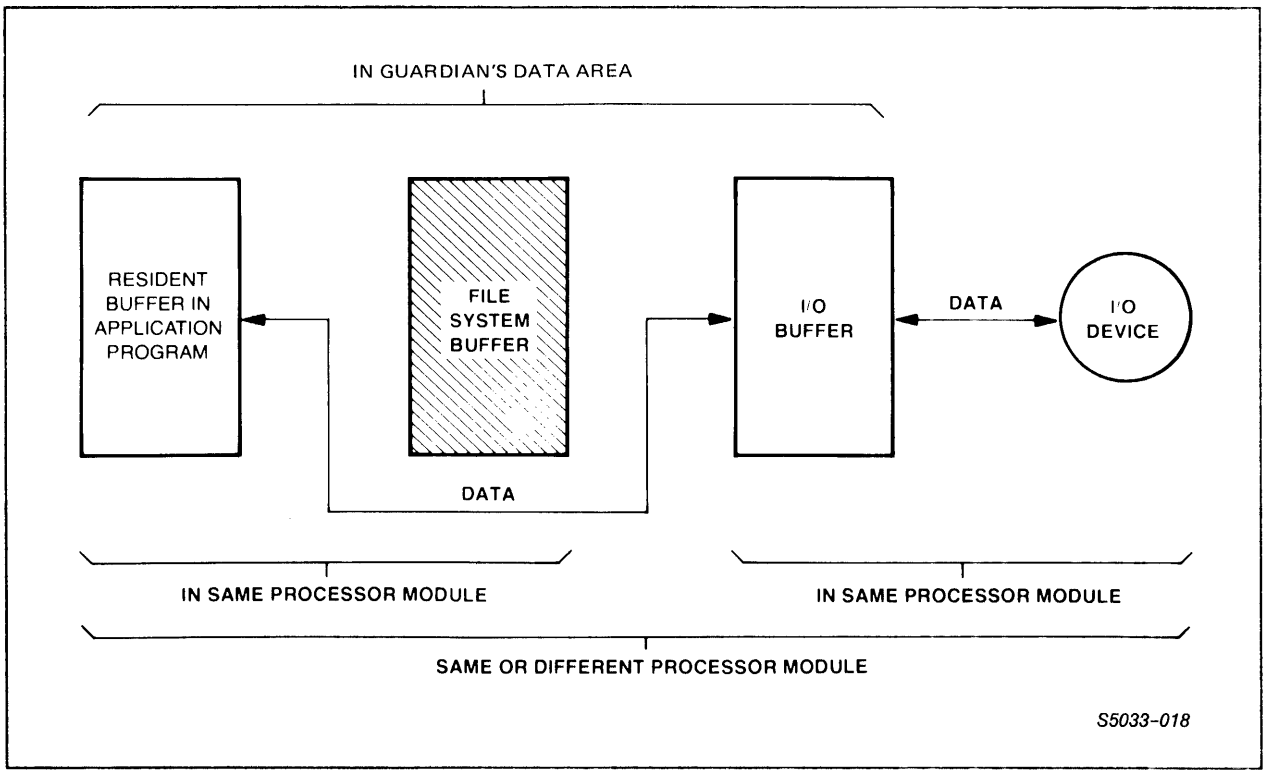


Figure 5-5. Resident Buffering

Resident buffers are specified on a file-by-file basis through bit <flags>.<6> of the OPEN procedure. If resident buffers are specified, the application process must make resident any buffers (arrays) used with the file in main memory. Additionally, the resident buffer in the application's data area must be addressable through the system data map. Both are done through a call to the process-control LOCKDATA procedure. LOCKDATA can be called only if the application process is executing in privileged mode (otherwise, an "instruction failure" trap will occur).

For example:

```
  INT .buffer[0:255]; ! application buffer to be
                      ! locked into memory.
  INT PROC priv^lockdata ( address , count , sysmap ) CALLABLE;
     INT address, count, sysmap;
    BEGIN
      RETURN LOCKDATA ( address , count , sysmap );
    END; ! priv^lockdata.
```

is a application procedure that executes in privileged mode. This is used instead of a direct call to LOCKDATA, so the process does not execute in privileged mode when not necessary.

Now, invoke LOCKDATA:

```
  n := priv^lockdata ( @buffer, 256, 1 );
```

This specifies that the physical pages where "buffer" is located are to be made main-memory-resident and are to be assigned to entries in the system data map. If the pages are successfully locked, 1 is returned in n.

Then call OPEN, specifying resident buffers:

```
  LITERAL res^buf := %1000;
    ...
  CALL OPEN ( file^name, file^number, flag LOR res^buf );
  IF < THEN ..     ! open failed.
```

A subsequent call to a file-system I/O procedure would then specify "buffer" in the procedure's <buffer> parameter. For example:

```
    ...
  buffer ':=' data FOR write^count;  ! move data into
                                     ! resident buffer.
  CALL WRITE ( file^number, buffer, write^count ); ! write it.
  IF < THEN ... ; ! error.
    ...
```

If a resident buffer is specified for the $RECEIVE file or a process file, the 12 words immediately preceding word[0] of the buffer must also be resident and available for use by the file system (that is, this space must be unused by the application process). For example:

```
  INT .buffer[-12:255]; ! application buffer to be locked ! in
                        memory.
```

LOCKDATA is invoked in this fashion:

```
  n := priv^lockdata ( @buffer[ -12 ], 264, 1 );
```

When you use resident buffering, these considerations apply:

- For each concurrent process running with resident buffering in any one CPU, at least 1K of system global data space must be left unassigned when generating the system (by SYSGEN).
- Although resident buffering is specified on a file-by-file basis, you can use the same resident buffer for several different files (if, of course, the structure of the program permits).
- It is not necessary to call LOCKDATA before OPEN is called. However, LOCKDATA must be called before the first I/O transfer (READ, WRITE, CONTROL, etc.) with a file is performed.
- The resident buffer is not used for accesses to structured ENSCRIBE files; it is for unstructured files only.

Adjustable Buffering (with DP2 Disc Processes Only)

The BUFFERSIZE attribute lets you define the internal buffer size to be used for access to an unstructured DP2 file. The buffer can be any valid DP2 block size--512, 1024, 2048, and 4096 bytes. You can set this attribute when the file is created and change it by either a call to SETMODE or use of the FUP command ALTER. If you specify an invalid size, the next higher valid size is used. The default buffer size is 4096, the highest possible. The buffer size you specify has no effect on the format of the data accessed by the user.

An appropriate buffer size lets the disc process use its fixed-length cache management scheme more efficiently. For instance, if some application usually accesses a DP2 unstructured file in 1024-byte quantities and on 1024-byte boundaries, the most useful buffer size would be 1024.

LOCKING FILES AND RECORDS

Access to a sharable file among two or more processes is coordinated by means of file locks, record locks, and key locks. A process requests a lock before performing a critical operation, to temporarily exclude other accesses. It releases the lock when a critical operation is completed, to allow access by other processes.

File-Locking

ENSCRIBE disc files can be locked and unlocked by the LOCKFILE and UNLOCKFILE procedures. Multiple processes accessing the same disc file can call LOCKFILE before performing a critical sequence of operations to that file. If the file is not currently locked, it becomes locked and the process continues executing. This prevents other accesses to the file until it is unlocked through a call to UNLOCKFILE. If the file is locked, then the action taken depends on the locking mode in effect at the time of the call (see "Interaction between File Locks and Record Locks" below).

If a process attempts to write to a locked file, the access is rejected with a "file is locked" error indication; if a process attempts to read from a locked file, it is suspended until the file is unlocked.

An alternate mode for file-locking can be specified via a call to the SETMODE procedure. Instead of suspending the caller to LOCKFILE if the requested file is locked, the lock request is rejected and the call to LOCKFILE completes immediately with a "file is locked" error indication. Moreover, if a process attempts to read from a locked file, the attempt is immediately rejected.

Record-Locking

Record-locking operates in essentially the same manner as file-locking, but it allows a greater degree of concurrent access to a single file than file-locking does.

Individual records of a file are locked by one of these procedures:

- The LOCKREC procedure locks the current record (as indicated by the last operation with the file).
- The READLOCK or READUPDATELOCK procedure locks the record to be read before reading it.

When a lock is requested for a record, if no other process has it locked, the lock is immediately granted. If the record or the file is locked, then the action taken depends on the locking mode in effect at the time of the call (see "Interaction between File Locks and Record Locks" below).

If a file lock is attempted on a file when one or more records are locked, the file lock must wait for all records to be unlocked before it will be granted. Similarly, a record lock

must wait if the file is currently locked.

Records are unlocked by one of these procedures:

- The UNLOCKREC procedure unlocks the current record.
- The UNLOCKFILE procedure unlocks all records in the file locked by the caller.
- The WRITEUPDATEUNLOCK procedure unlocks the record after writing it.

If a record is deleted, it is automatically unlocked. If a record is deleted from a file audited by the Transaction Monitoring Facility (TMF), the lock is not relinquished until the transaction is committed or aborted. See the GUARDIAN Operating System Programmer's Guide or the PATHWAY SCREEN COBOL Reference Manual for details.

Record-locking allows the maximum concurrency of access to a file while still guaranteeing the integrity of the file's contents when it is to be simultaneously updated by more than one process. However, for complex updating of a file involving many records, record-locking may not be desirable, because of the amount of system processing required or because it increases the possibility of "deadlock" (see "Deadlock" below). In such cases, file-locking may be preferable.

In a NonStop 1+ system, the maximum number of concurrent locks depends on how much control-block space is available, up to a maximum of 922 key locks or 1808 record locks. In a NonStop system with DP1 disc processes, no more than 3000 concurrent record locks or 2000 key locks can be pending on a given file. If the file is partitioned, the limit applies to each partition, rather than to the entire file, because each partition has its own file control block (FCB).

In a NonStop system with DP2 disc processes, no more than 2000 locks (of all kinds) can be held concurrently by any user--that is, by any file opener or any TMF transaction identifier.

If the maximum is reached and an additional lock request is made, the lock request will be rejected with <error> = 35 ("unable to obtain I/O process control block") on a NonStop system or <error> = 32 ("unable to obtain main memory space for a control block") on a NonStop 1+ system.

When a process reads a file that was opened with sequential block-buffering, the process ignores all record locks (although it does honor file locks). The File Utility Program (FUP) command COPY, for example, uses sequential block-buffering and therefore it can read locked records.

Key-Locking (DP1 Only)

Key locks are used only for deleted records in audited DP1 files, because there is no address to define a record lock. TMF implicitly sets these locks when records are deleted.

The key-lock entry size varies with the key length, unlike record locks, whose entries are of a fixed size. This means you are not always guaranteed space for 2000 key locks; if the keys being locked are long, fewer key locks will be allowed. Therefore, the application designer should try not to delete large numbers of records in a single transaction.

Locking Modes

Locks are granted on an file-opening (<file number>) basis. Therefore, if a process has multiple openings of the same file, a lock through one file number excludes access to the file through other file numbers.

Two locking modes are available. The locking mode determines the action taken if the file or record is already locked when a request is made to lock it.

- In the default locking mode, if a process requests to lock or read a locked record (that is not locked by the <file number> supplied in the call), that process is suspended until the file or record becomes unlocked.

- In the alternate locking mode, if a process requests to lock or read a locked record (that is not locked by the <file number> supplied in the call), that request is immediately rejected with a "file/record is locked" error indication (<error> = 73), so the requesting process can take alternative action.

The locking mode is specified via <function> 4 of the SETMODE procedure.

In either mode, if a control or write request is made and the requested record is locked but not through the <file number> supplied in the call, the call is rejected with a "file/record is locked" error indication (<error> = 73).

Interaction between File Locks and Record Locks

This subsection applies only if the default locking mode is in effect.

With the DP1 disc process, mixed record-locking and file-locking in a given file is not supported. Record locks cannot be granted while the file is locked. With DP2, a user (file-opener or TMF transaction identifier), having locked a file, still can lock a record in that file separately.

For a file having one or more pending lock requests, there is a queue of file-locking requests. Attempting to read from a locked file, when the file is not locked through the file number supplied in the call, queues the read request with the file-locking requests. When the current lock is cleared (by means of a call to the UNLOCKFILE procedure), the request at the head of the file-locking queue is granted. If the request is a lock request, the lock is granted and the request continues processing; if it is a read request, the file is read.

Similarly, for a record having one or more pending lock requests, there is a queue of record-lock requests. Attempting to read from a locked record, when the record is not locked through the file number supplied in the call, queues the read request with the record-locking requests. When the current lock is cleared, the system grants the request at the head of the locking queue for that record. If the request is a lock request, the lock is granted and the processing continues; if it is a READ request, the record is read.

A call to LOCKFILE is not equivalent to locking all records in a file; for instance, locking all records would still allow someone else to insert new records but file-locking would not.

With the DP2 disc process, locking requests do not wait behind other locks held by the same user. If a user holds record locks and later requests a file lock, and no other user holds record locks in that file, the record locks are given up and replaced by the file lock.

With the DP1 disc process, if a user requests a file lock while any records in the file are locked, the request is queued behind the record locks. If a user requests a record lock in a file it has already locked, that request is usually queued behind any file locks for the file.

The exception to the DP1 queuing of record locks is: if a user has one or more records locked, then requests another record lock for that file, the record lock will preempt any pending (but not yet granted) file locks for that file (the request will not preempt other record locks for the same record). This exception

minimizes the possibility of a deadlock condition occurring, as illustrated below:

```
        User A                  User B                  User C
  ---------------------   -----------------   -------------------
     .                       .                       .
     .                       .                       .
  LOCKREC: $A.B.C,rec 1      .                       .
  (lock granted)             .                       .
     .                    LOCKFILE: $A.B.C           .
     .                    (lock queued)              .
     .                                          LOCKREC: $A.B.C,rec 12
     .                                             (lock queued)
  LOCKREC: $A.B.C,rec 12
  (lock granted)
     .
     .
```

Note that a deadlock condition occurs in this situation:

1. A process opens a certain file more than once, with file numbers 1, 2, ..., n, using the default locking mode.

2. The process locks file number 1.

3. The process calls READ or READUPDATE, using file number 2.

Now the READ must wait until the file lock is removed, so the process is suspended. The suspended process, however, cannot remove the lock, so a deadlock condition exists. If this process uses the alternate locking mode, the READ request would be rejected and processing would continue. Deadlocks are discussed further in the next subsection.

Deadlock

One problem that can occur when multiple processes require multiple record locks or file locks is a deadlock condition. An example of deadlock is:

```
        User A                  User B
  -----------------       -----------------
  LOCKREC: record 1       LOCKREC: record 2
     .                       .
     .                       .
  LOCKREC: record 2       LOCKREC: record 1
```

Here, user A has record 1 locked and is requesting a lock for record 2, while user B has record 2 locked and is requesting a lock for record 1.

One possible way to avoid deadlock is to always lock the records in the same order. Thus, the situation described above would never happen if each user requested the lock to record 1 before it requested the lock to record 2.

Since it is sometimes impossible for an application program to know in which order the records it must lock are going to be encountered, another solution is offerred. For updates to single records of the file, no special processing need be done. For an update involving two or more records, however, the solution is to first lock some designated common record, and then lock the necessary data records. This prevents deadlock among those users requiring multiple records, since they must first gain access to the common record, but still allows maximum concurrency and minimum overhead for accessors of single records.

Record-Locking with Unstructured Files

ENSCRIBE permits record-locking in unstructured files, so applications can define their own file structures and still use the capabilities of record-locking.

Record-locking with unstructured files is accomplished by positioning the file to the relative byte address of the record to be locked, then locking the address through the LOCKREC, READLOCK, or READUPDATELOCK procedure. Any other user attempting to access the file at a point beginning at exactly that address will see the address as being locked (the action will then be appropriate for the current locking mode).

Unstructured files are unlocked in the same way as structured files.

TMF Locking Considerations

In a system using the Transaction Monitoring Facility (TMF), a transaction must lock all records that it updates, either on a record-by-record basis or for an entire file at a time. For server processes to change a TMF-audited data base, TMF imposes certain record-locking rules to prevent transactions from reading uncommitted changes of other concurrent transactions:

- Whenever a transaction inserts a new record into an audited file, that record is locked automatically.
- Before a process can successfully change or delete an existing record in an audited file, it must previously have locked the record or file. When a transaction deletes a record from an

audited file, TMF sets a key lock based on the deleted record's primary-key value. For the duration of the record-deleting transaction, the key lock effectively prevents any other transaction from inserting a record having the same key value as the deleted record.

- All locks on audited files are held until the transaction completes (that is, until it either is committed or aborts and is backed out).
- At the discretion of the programmer, all records read and not changed, but used by a transaction in producing its output, should be locked. Following this rule guarantees that all reading operations are repeatable.

All but the last rule are enforced by TMF on audited files.

A TMF transaction is begun by a call to BEGINTRANSACTION and terminated by a call to ENDTRANSACTION.

Figure 5-6 illustrates (1) how processes can acquire locks and update audited files and (2) when TMF will release the locks.

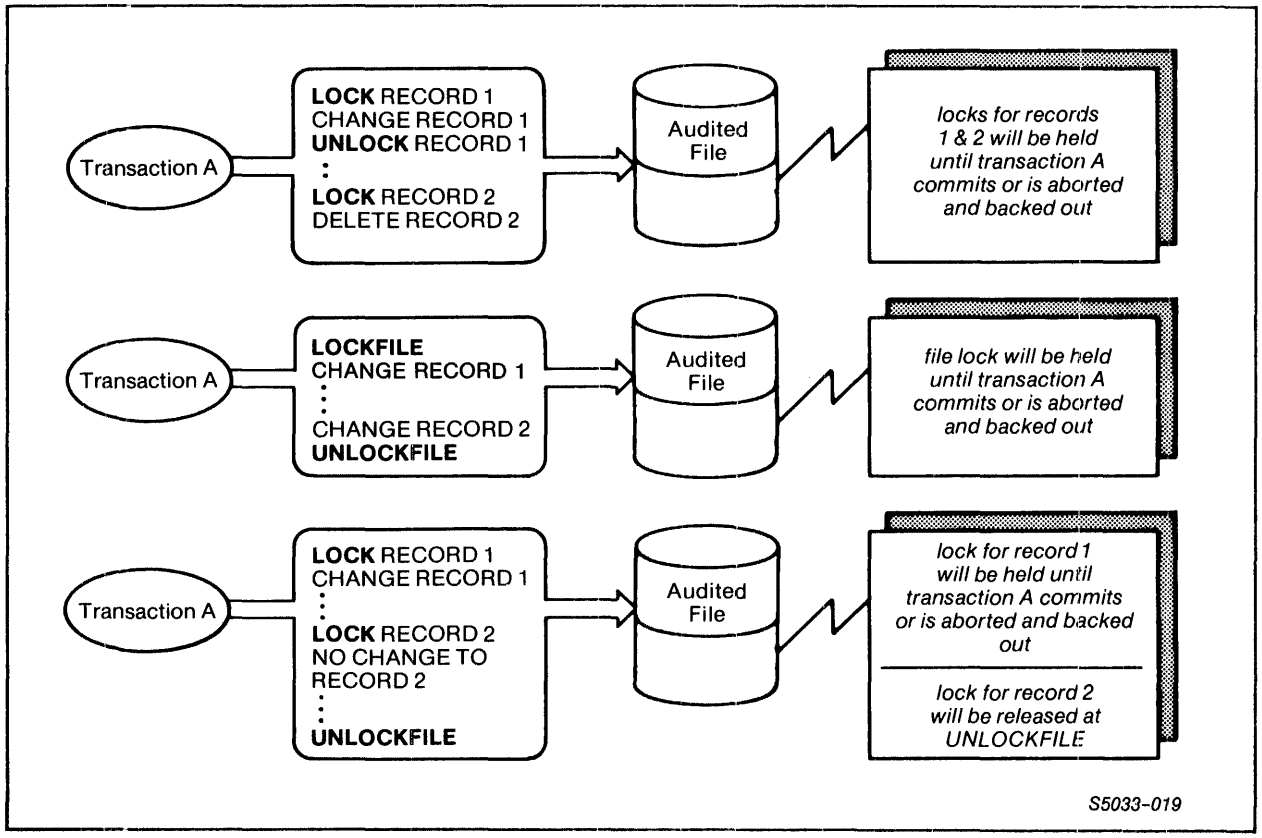


Figure 5-6. Record-Locking for TMF

If the whole set of current active transactions tries to acquire too many locks (see "Record-Locking" above), the attempt will be rejected with <error> = 35 ("unable to obtain I/O process control block") on a NonStop system or <error> = 32 ("unable to obtain main memory space for a control block") on a NonStop 1+ system.

The file lock or record locks are owned by the current-transaction identifier of the TMF process that issued the lock request. For example, a single transaction can send requests to several servers or multiple requests to the same server class. In this situation, where several processes share a common transaction identifier and the locks are held by the same transaction identifier, the locks do not cause conflict among the processes participating in the transaction (see Figure 5-7).

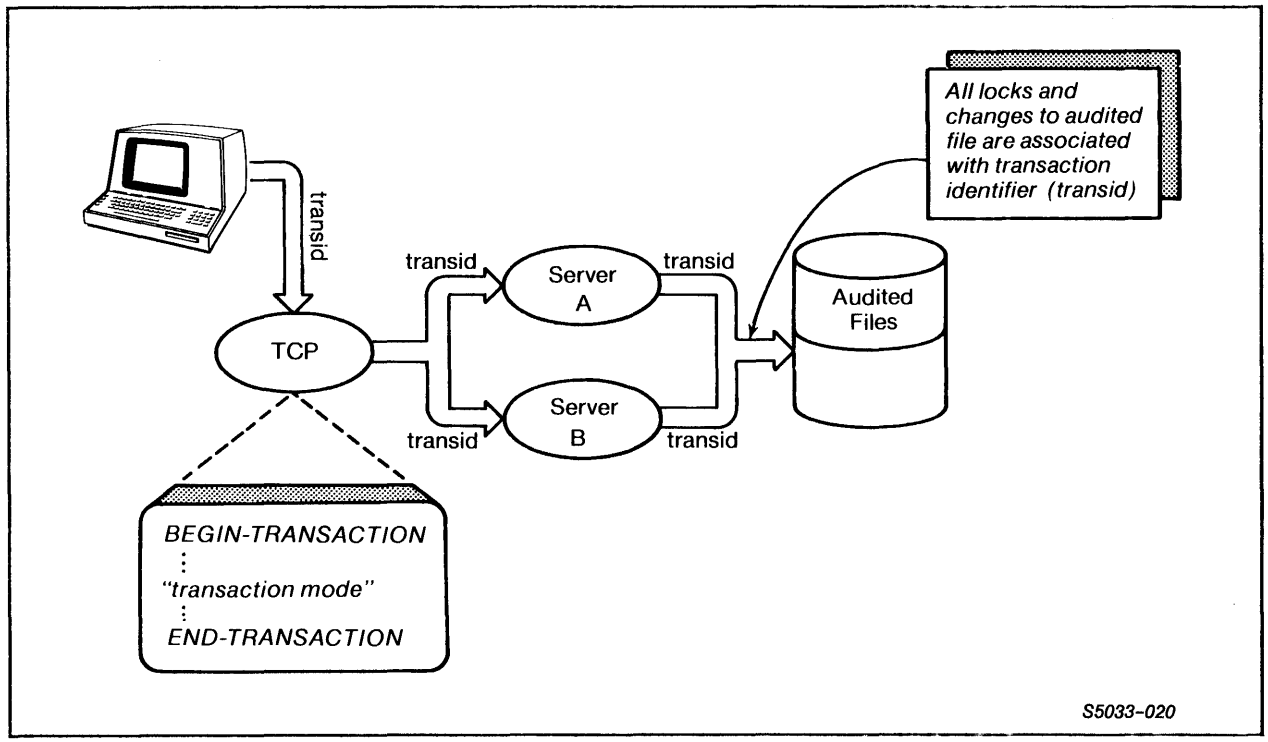


Figure 5-7. Record-Locking by Transaction Identifier

Figure 5-7 illustrates these principles:

- The terminal control process (TCP) interprets BEGIN-TRANSACTION and obtains the transaction identifier before requesting data-base activity from the servers.
- The transaction identifier is transmitted to the servers in the request message and any disc activity performed by the servers is associated with the transaction identifier.

- The transaction identifier owns the lock(s); all servers that acquired the same transaction identifier can read, lock, add, delete, and change records in the audited files. For example: server A can read and lock a record and server B can read or change the same record, if both servers A and B have the same current-transaction identifier.

REPEATABLE READS. Generally, a TMF transaction should lock any data it reads and uses in producing its output, regardless of whether it modifies the data. Following this rule guarantees that all of a transaction's reading operations are repeatable and that data on which the transaction depends does not change before the transaction is committed.

OPENING AUDITED FILES--ERRORS. In TMF, because locks are owned by the transaction identifier instead of the process identifier or the identifier of the file opener, they can persist longer than the opener process. This means that even if a file has been closed by all its openers, the disc process will keep it effectively open until all transactions owning locks in the file have ended or have been aborted and backed out.

For files with pending transaction locks, these types of errors are possible:

- Attempting to open an audited file with exclusive access will fail with <error>=12 ("file in use"), regardless of whether openers of the file exist.

- FUP operations requiring exclusive access such as PURGE and PURGEDATA will fail. PURGE will fail with file error 12 and PURGEDATA will fail with file error 80.

Additionally, error 80 ("invalid operation on audited file") will be returned for attempting to open a file if:

- It is a nonaudited file whose automatically updated key file that is audited.

- It is an audited file whose automatically updated key file that cannot be opened or is not audited.

- It is a structured audited file with unstructured access.

- It is an audited, partitioned file having a nonaudited secondary partition.

- It is a nonaudited, partitioned file having an audited secondary partition.

READING DELETED RECORDS. If transaction T1 deletes a record and another transaction T2 attempts to read the same record while T1 is still active, then:

- If T2's request is from the READ procedure after exact positioning, error 1 ("end-of-file") will be returned.
- If T2's request is from the READUPDATE procedure, error 73 ("file or record locked") will be returned (in alternate locking mode) or the request will wait for T1 to complete (in default locking mode).

BATCH UPDATES. When programming for batch-updating of audited files, you should either have the transaction lock an entire file at a time by using the LOCKFILE procedure or carefully keep track of the number of locks held. If you do not use LOCKFILE, TMF sets these implicit locks:

- When a new record is inserted in an audited file, TMF implicitly locks that record.
- When a record is deleted from an audited file, TMF implicitly retains a lock on the key of that record.

These locks are not released until the transaction is committed or is aborted and backed out. This means that transactions doing batch updates to audited files, if they involve deleting, updating, or inserting a large number of records, can seek too many locks. (The maximum number of locks that can be acquired for each DP1 file or by each DP2 transaction is specified in the "Record-Locking" subsection above.) This situation will return <error> = 35 ("unable to obtain I/O process control block") on a NonStop system or <error> = 32 ("unable to obtain main memory space for a control block") on a NonStop 1+ system.

If a TMF transaction calls LOCKFILE for a primary file, LOCKFILE is automatically applied to any associated alternate-key files. This prevents primary-file updates from causing the alternate-key files to obtain record or key locks.

OTHER CONSIDERATIONS FOR BOTH STRUCTURED AND UNSTRUCTURED FILES

This subsection describes data purging, WRITE verification, file refreshing, and allocation and deallocation of extents.

Purging Data

Either the File Utility Program's (FUP's) PURGE command or the PURGE procedure can logically remove all data from a file by removing the file from the disc directory. The file data are not overwritten or erased, but rather pointers are changed to show the data to be absent.

Later, if that space is re-allocated for another file, the new file's owner may be able to read the logically purged data. For security reasons, you may want to actually erase the data. If you have set the CLEARONPURGE flag for a file, using either function 1 of the SETMODE procedure or the FUP command SECURE, all data will be physically erased (overwritten with zeros) when the file is purged. (See the "SETMODE and SETMODENOWAIT Functions" table in the System Procedure Calls Reference Manual.

Also, either the FUP command PURGEDATA or the CONTROL procedure's "purge data" operation can logically remove all data from a nonaudited file by resetting the file's current-record, next-record, and end-of-file (EOF) pointers to relative byte 0 and updating the EOF pointer in the file label on disc. CLEARONPURGE has no effect after PURGEDATA, however.

For example, this CONTROL call would logically purge all data from file <file^a>.

```
  ...
LITERAL purgedata = 20;
  ...
CALL CONTROL ( file^a, purgedata );
IF < THEN...
```

This CONTROL operation can be used in conjunction with the CONTROL procedure's "allocate/deallocate" operation to deallocate all of a file's extents:

```
LITERAL alloc^op  = 21,
        dealloc   =  0;

CALL CONTROL ( file^a, purgedata );
IF < THEN ...
CALL CONTROL ( file^a, alloc^op, dealloc );
IF < THEN ...
```

sets the EOF pointer to relative byte 0 then deallocates all extents.

WRITE Verification

Using the "verify write" operation ensures the integrity of each WRITE operation to a disc file. The disc-controller hardware makes a byte-by-byte comparison of the just-written data on disc with the corresponding data in the controller's memory. Note, however, that this requires an additional disc revolution. The "verify write" option is enabled by the SETMODE procedure (<function> = 3); the default setting disables it. Also, you can enable this option when you open a DP2 file.

Refreshing

The information in an open file's FCB, such as the end-of-file (EOF) pointer, is kept in main memory. To maximize performance, the EOF pointer is normally written to the file's disc label only when needed, as described in the "End-of-File Pointer" subsection above.

Although refreshing the file's disc label (under these conditions only) maximizes system performance, certain considerations should be taken into account:

- If an open file is backed up, the file label copy on tape does not reflect the actual state of the file. An attempt to restore such a file will result in an error.
- If the system is shut down (each processor module has been reset) while a file is open, the file label on disc will not reflect the actual state of the file.
- If a total system failure occurs (such as that caused by a power failure that exceeds the limit of memory battery backup) while a file is open, the file label on disc will not reflect the actual state of the file.

An autorefresh option is available, indicated by setting <file type>.<10> of the CREATE procedure, that causes the file label to be written to disc each time the end-of-file pointer is advanced. You can also use the FUP command ALTER...REFRESH to specify the autorefresh option (see the the descriptions of the File Utility Program (FUP) in the GUARDIAN Operating System Utilities Reference Manual and the GUARDIAN Operating System User's Guide). REFRESH is always ON for DP2 key-sequenced files.

However, the additional I/O caused by the REFRESH ON option can decrease processing throughput significantly. For applications (other than DP2 key-sequenced files) that cannot afford this overhead, the file label on disc can be forced to represent the actual state of a file through periodic use of the REFRESH

procedure or the equivalent Peripheral Utility Program (PUP) REFRESH command. Execution of REFRESH writes the information contained in any file control blocks (FCBs) to the file labels on the associated disc volume.

REFRESH is useful before backing up a file that is always open--for example, where the application is always running. At some point during the day when the system is quiescent (no transactions are taking place), issue a REFRESH command for all volumes in the system. Then, when the files are backed up, the file labels on backup tape will represent the actual states of the files backed up.

You can also use REFRESH before a total system shutdown to ensure that the file labels on disc will represent the actual states of files on disc.

To minimize the effect of a total system failure, have an application process call the REFRESH procedure periodically (say, every ten minutes). Note that this is useful only when a power failure occurs that exceeds the limit of memory battery backup; therefore, it is necessary only where the computer site is susceptible to frequent and severe power outages.

For DP2 files, using the ALL option with REFRESH ensures that FCBs and all file blocks are flushed.

Programmatic Extent Allocation

An application process can cause the file system to allocate one or more file extents in an open file by means of the CONTROL procedure's "allocate/deallocate" operation.

For example, to allocate all 16 extents in a newly created DP1 file, the file is opened then CONTROL is called, as shown here:

```
LITERAL alloc^op = 21,
        max^ext  = 16;
  ...
CALL CONTROL ( file^a, alloc^op, max^ext );
IF < THEN ... ! extent allocate error.
  ...
```

This allocates all 16 extents of "file^a".

Extent Allocation Errors

Two errors are associated with allocating disc extents: 43 ("unable to obtain disc space for file extent") and 45 ("disc file full").

The next example shows both kinds of error. A file is created with an extent size of 2048 bytes, then repetitive WRITEs of 400 bytes are executed to the file:

```
loop: CALL WRITE ( file^a, buffer, 400, number^written );
      IF < THEN
        BEGIN
          CALL FILEINFO ( file^a, error );.
          ...
        END
      ELSE GOTO loop;
```

The first five WRITEs are successful ("number^written" = 400), but the sixth fails after transferring 48 bytes of "buffer" to the disc ("number^written" = 48). If all disc space has been allocated to other files, the <error> returned by FILEINFO is 43 ("out of disc space"). If the current extent is the last one permitted in the file (extent number 15 in a DP1 file), then the <error> returned by FILEINFO is 45 ("disc file full"). This situation is illustrated in Figure 5-8.

Note that <error> = 43 can also occur when allocating extents via the CONTROL procedure's operation 21 ("allocate/deallocate").

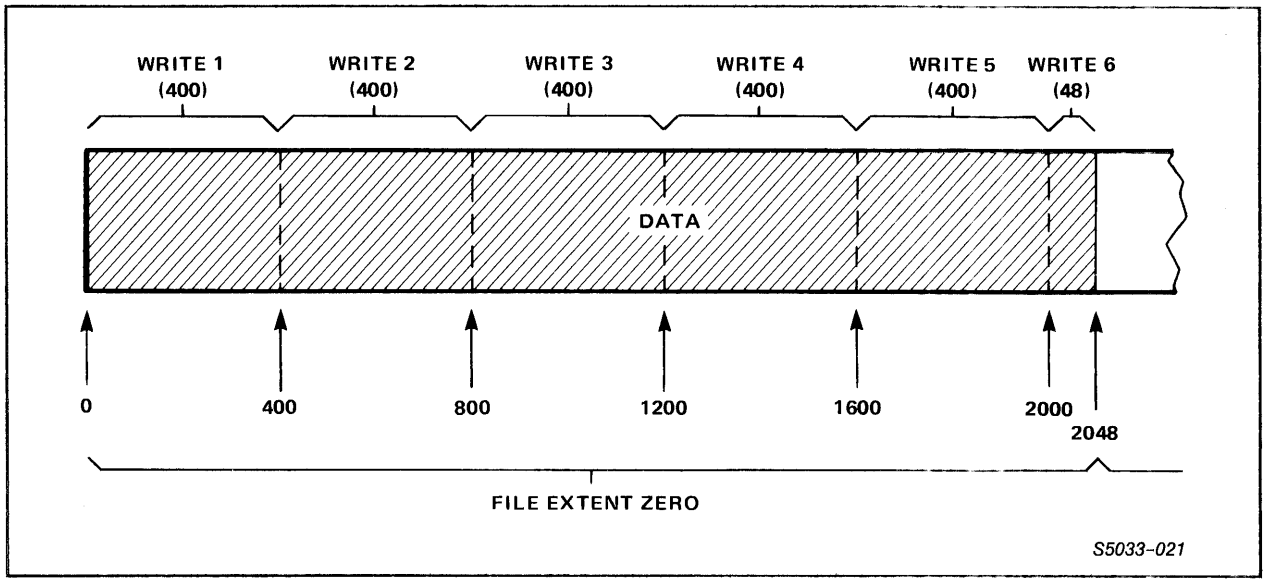


Figure 5-8. Example Showing Extent-Allocation Error

Programmatic Extent Deallocation

An application process can cause the file system to deallocate any file extents past the extent where the end-of-file pointer is pointing, by means of the CONTROL procedure's "allocate/deallocate" operation.

For example, to deallocate any unused extents in a file, the file is opened, then CONTROL is called, as shown here:

```
LITERAL alloc^op = 21,
        dealloc  =  0;
  ...
CALL CONTROL ( file^a, alloc^op, dealloc );
  ...
```

This deallocates any extents past the end-of-file extent.

DISC CONTROL AND SETMODE OPERATIONS

Complete descriptions of the CONTROL operations and SETMODE and SETMODENOWAIT functions are given in the System Procedure Calls Reference Manual.

ERRORS AND ERROR RECOVERY

The file system produces a number of messages indicating errors or other special conditions. These messages can occur during execution of almost any user application or Tandem-supplied program, since most programs use the file system.

An error number is associated with the completion of each procedure call. (See "Characteristics of ENSCRIBE Procedure Calls" in Section 3.)

Error Categories

File-system errors are grouped into three major categories, as described in Table 3-2. These categories are:

- Errors reserved for process application-dependent use (error numbers 300 through 511)
- Errors encountered by standard operations (error numbers 10 through 255)
- Warnings (error numbers 1 through 9)

A "fourth category" is <error> = 0, which is "no error."

Many of the file-system errors imply that invalid parameters were supplied to the file-system procedures or that illegal operations were attempted. These could be considered programming errors. Other types of errors imply that the system is not being operated properly. And other types are simply informational messages informing the application about a particular device-oriented problem.

Errors occuring during disc file access can be separated into these categories: communication-path errors, data errors, device-operation errors, and failure of the primary application process.

Communication-Path Errors

A communication-path error is a failure of a processor module, I/O channel, or disc-controller port that is part of the primary path to disc device. For errors of this type, the file system automatically switches to the alternate path and completes the I/O operation if a synchronization depth greater than zero was specified when the file was opened. Therefore, if an error >= 200 is returned to the application program, the disc device is no longer accessible.

Data Errors

Data errors are error numbers 50 through 59, 120 through 139, and 190 through 199. The file system automatically retries operations associated with this type of error. Therefore, if one of these errors is returned, all or part of the file can be considered invalid. Error 120 ("data parity error") indicates a hardware error. A bad track on a NonStop 1+ system, or a bad sector on a NonStop system, can be assigned to an alternate track or sector through use of the Peripheral Utility Program (PUP) command SPARE.

Device-Operation Error

Device-operation errors are error numbers 60 through 69 and 103. None of these errors are retried by the file system. Errors 60-69 indicate that the device has been deliberately been made inaccessible and, therefore, the associated operation probably should not be retried. Error 103 occurs if the entire system has experienced a power failure and that the disc is in the process of becoming ready. Therefore, an operation associated with error 103 should be retried periodically.

Failure of the Primary Application Process

A failure of the primary application process is not a disc error in the strictest sense. Rather, this is a failure of the processor module where the primary process of a primary/backup process pair is executing. Operations associated with this type of failure must be retried by the backup application process when it takes over the applications work. The GUARDIAN Operating System Programmer's Guide discusses recovery from this type of error, under "Checkpointing."

Error-Recovery Routines

When writing error-recovery routines, the programmer must consider the type of device (disc, magnetic tape, line printer, etc.); the type of error (whether it is recoverable programmatically); and the number of "no-wait" operations outstanding when the error is detected.

If a disc file is opened with a <sync depth> greater than or equal to 1, all recoverable errors, including path errors, are automatically retried by the file system.

When using "no-wait" I/O and executing more than one concurrent operation to the same file, it is quite possible for one operation to fail, but subsequent operations to succeed, as illustrated here:

Three "no-wait" WRITE operations are initiated to a line printer:

- WRITE "no-wait" 1 initiated to printer
  print: "line one"

- WRITE "no-wait" 2 initiated to printer
  print: "line two"

- WRITE "no-wait" 3 initiated to printer
  print: "line three"

Then the three WRITE operations are completed with calls to AWAITIO:

- AWAITIO 1 indicates "no-wait" 1 succeeded (line printed)

- AWAITIO 2 indicates "no-wait" 2 failed (line not printed)

- AWAITIO 3 indicates "no-wait" 3 succeeded (line printed)

The three "no-wait" WRITE operations generate this output:

```
"line one"
"line three"
```

Note that "line two" is missing. When order is important, you should not permit concurrent operations on the same file.

Error Considerations for DP1 Key-Sequenced Files

DP1 users are cautioned that if a key-sequenced file is opened with a <sync depth> of zero, a failure occuring while a record is being inserted or updated may leave the structure of the file in an indeterminate state (and therefore inaccessible).

Error Considerations for Files with Alternate Keys

Users are cautioned that if an application process opens a file having alternate keys with a <sync depth> of zero or opens with a nonzero <sync depth> but does not have a backup to complete an operation in case of a failure, a failure occurring while a record is being inserted or updated will have indeterminate results. It is possible in such cases for the insertion (or update) of the primary record to be successful but the insertion (or update) to the alternate-key file to have not been made. In this instance both files will appear valid (their structures are intact); however, there will be no alternate-key reference to the primary record.

Error Considerations for Partitioned Files

Each partition of a file can get errors apart from the file's other partitions. This is especially significant for errors 42 through 45, which pertain to disc-space allocation. You may be able to avoid these errors by using FUP to alter the size characteristics of the partition where the error occurred.

In any case, after a CCL (or possibly CCG) return from a file-system procedure call, the file-system error number is obtained by calling the FILEINFO procedure; the partition number of the partition in error is obtained by calling the FILERECINFO procedure. The volume name of the partition in error can be read by examining the file's partition-parameter array (either programmatically or via FUP).

ACTION OF CURRENT KEY, KEY SPECIFIER, AND KEY LENGTH

To briefly describe the basic file-system operations and their relationships to file-currency information, we first define some variable names and some functions. We then use those variable names and functions to write miniprograms defining the OPEN, KEYPOSITION, POSITION, READ, READUPDATE, WRITEUPDATE, and WRITE procedures.

First, we define some variables:

- CKV = <current key value>
- CKS = <current key specifier>
- CKL = <current key length>
- CMPL = <comparison length>
- MODE = <positioning mode>: approximate = 0
generic = 1
exact = 2
- primary = 0
- next = true if the next record in sequence is to be the reference
- rip = relative file insertion pointer
- present = true if parameter is supplied
- keyseq = file type 3
- entryseq = file type 2
- relative = file type 1

Next, we define some functions:

- keyfield (record, specifier)

  returns the value of the "specified" key field in the record. If the file is not key-sequenced and specifier = 0, then a

<record specifier> is returned.

- keylength (record, specifier)

  returns the length of the "specified" key field in the record. If record = 0, this returns the defined key-field length.

- find (mode, specifier, key value, comparison length)

  returns the position of the first record in the file according to mode, specifier, key value, and comparison length.

  If mode = 0 (approximate), positioning is to the first record whose key field, as designated by the <key specifier>, is greater than or equal to the <key value>. If no such record exists, an end-of-file indication is returned.

  If mode = 1 (generic), positioning is to the first record whose key field, as designated by the <key specifier>, contains a value equal to <key> for <comparison length> bytes. If no such record exists, an end-of-file indication is returned.

  If mode = 2 (exact), positioning is to the first record whose key field, as designated by the <key specifier>, contains a value of exactly <comparison length> bytes and is equal to <key>. If no such record exists, an end-of-file indication is returned.

- find^next (mode, specifier, key value, comparison length)

  returns the position of the next record in the file according to mode, specifier, key value, and comparison length.

  If mode = 0 (approximate), positioning is to the next record.

  If mode = 1 (generic), positioning is to the next record. If the key field designated by the <key specifier> does not equal <key> for <comparison length> bytes, an end-of-file indication is returned.

  If mode = 2 (exact), an end-of-file indication is returned.

- insert (key value, key length);

  returns the position where a record is to be added, according the specified key value and key length. If a record already exists at the indicated position, a "duplicate record" indiction is returned. For relative and entry-sequenced files, a key value of "-1D" returns the end-of-file position and a key value of "-2D" returns the position of the first available record.

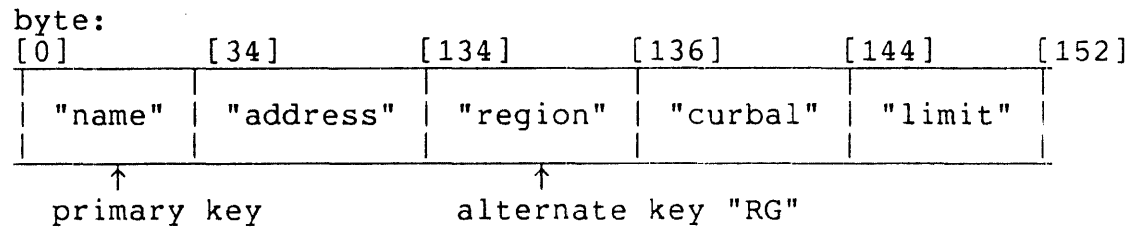
Now we can use the variables and functions defined above to show miniprograms that briefly describe the basic file-system operations and their relationships to file-currency information:

```
OPEN:
  CKS := primary;
  if keyseq then CKL := CMPL := 0
  else
    begin
      CKL := 4;
      CKV := rip := 0D;
    end;
  MODE := approx;
  next := false;

KEYPOSITION:
  CKV := rip := <key>;
  CKS := if present then <key specifier> else primary;
  CKL := CMPL := if present then <comparison length>
                 else keylength(0, CKS);
  MODE := if present then <positioning mode> else approx;
  next := false;

POSITION:
  CKV := rip := <record specifier>;
  CKS := primary;
  CMPL := CKL := 4;
  MODE := approx;
  next := false;

READ:
  position := if next then find^next(MODE,CKS,CKV,CMPL)
              else find (MODE,CKS,CKV,CMPL);
  if <error> then return;
  record := file[position];
  CKV := keyfield (record,CKS);
  CKL := keylength(record,CKS);
  next := true;

READUPDATE:
  position := find(exact,CKS,CKV,CKL);
  if <error> = 1 then <error> := 11; if <error> then return;
  record := file[position];

WRITEUPDATE:
  position := find(exact, CKS, CKV, CKL);
  if <error> = 1 then <error> := 11; if <error> then return;
  if <write count> = 0 then
    if entryseq then begin <error> := ##; return; end;
    else delete the record
  else file[position] := record;
```

```
WRITE:
  if keyseq then
    begin
      position := insert(keyfield(record,primary),
                         keylength(record,primary));
      if <error> then return;
      file[keyposition] := record;
    end;
  if relative then
    begin
      if CKS then begin <error> := ##; return; end;
      if rip <> -2D and rip <> -1D and next then rip := rip+1;
      position := insert(rip,4);
      if <error> then return;
      file[position] := record;
      CKV := keyfield(record,primary);
      next := true;
    end;
  if entryseq then
    begin
      if CKS then begin <error> := ##; return; end;
      position := insert(-1D,4); ! end-of-file
      file[position] := record;
      CKV := keyfield(record,primary);
      next := true;
    end;
```

ACCESS EXAMPLES

All examples in this subsection use this CUSTOMER record definition:




```
INT .cust[0:75];                          ! customer record.

STRING
    .scust := @cust '<<' 1;               ! byte addressable.

FIXED(2)
    .fcust := @cust;                      ! fixed addressable.

DEFINE
     cust^len          =  152#,           ! customer-rec. length.
     cust^name         =  scust#,         ! name field.
     cust^name^len     =  34#,            ! name-field length.
     cust^address      =  scust[34]#,     ! address field.
     cust^address^len  =  100#,           ! address-field length.
     cust^region       =  scust[136]#,    ! region field:
                                          !   NO = northern,
                                          !   SO = southern,
                                          !   EA = eastern,
                                          !   WE = western.
     cust^region^len   =  2#,             ! region-field length.
     cust^curbal       =  fcust[17]#,     ! current balance.
     cust^limit        =  fcust[18]#;     ! credi- limit field.
```

The contents of the CUSTOMER file are:

name	address	region	curbal	limit
ADAMS	MIAMI, FL.	SO	0000.00	0500.00
BROWN,A	REEDLEY, CA.	WE	0256.95	0300.00
BROWN,B	BOSTON, MA.	EA	0301.00	1000.00
EVANS	BUTTE, MT.	WE	0010.00	0100.00
HARTLEY	CHICAGO, IL.	NO	0433.29	0500.00
JONES	DALLAS, TX.	SO	1234.56	2000.00
KOTTER	NEW YORK, NY.	EA	0089.00	0500.00
RICHARDS	MINNI, MN.	NO	0000.00	0500.00
ROGERS	BOISE, ID.	WE	1024.00	1500.00
SANFORD	L.A., CA.	WE	0301.00	1000.00
SMITH	DAYTON, OH.	NO	0010.00	0500.00

Example 1. Action of Current Position

- Position via primary key "ROGERS"

```
key := " ";                                  ! blank the
key[1] ':=' key FOR cust^name^len - 1; ! key.

key ':=' "ROGERS";
CALL KEYPOSITION ( cust^file^number, key);
```

ADAMS	MIAMI, FL.	SO	0000.00	0500.00	
BROWN,A	REEDLEY, CA.	WE	0256.95	0300.00	
BROWN,B	BOSTON, MA.	EA	0301.00	1000.00	
EVANS	BUTTE, MT.	WE	0010.00	0100.00	
HARTLEY	CHICAGO, IL.	NO	0433.29	0500.00	
JONES	DALLAS, TX.	SO	1234.56	2000.00	
KOTTER	NEW YORK, NY.	EA	0089.00	0500.00	
RICHARDS	MINNI, MN.	NO	0000.00	0500.00	
ROGERS	BOISE, ID.	WE	1024.00	1500.00	<---+
SANFORD	L.A., CA.	WE	0301.00	1000.00	<+ |
SMITH	DAYTON, OH.	NO	0010.00	0500.00	| |

next of subset |

current/

- Position via alternate key "RG" - "NO":

```
key ':=' "NO";
CALL KEYPOSITION ( cust^file^number, key, "RG");
```

BROWN,B	BOSTON, MA.	EA	0301.00	1000.00	
KOTTER	NEW YORK, NY.	EA	0089.00	0500.00	
HARTLEY	CHICAGO, IL.	NO	0433.29	0500.00	<---+
RICHARDS	MINNI, MN.	NO	0000.00	0500.00	<+ |
SMITH	DAYTON, OH.	NO	0010.00	0500.00	| |
ADAMS	MIAMI, FL.	SO	0000.00	0500.00	| |
JONES	DALLAS, TX.	SO	1234.56	2000.00	| |
BROWN	REEDLEY, CA.	WE	0256.95	0300.00	| |
EVANS	BUTTE, MT.	WE	0010.00	0100.00	| |
ROGERS	BOISE, ID.	WE	1024.00	1500.00	| |
SANFORD	L.A., CA.	WE	0301.00	1000.00	| |

next of subset |

current/

Example 2. Approximate Subset by Primary Key After OPEN

```
INT .cust^file^name[0:11],
     cust^file^number,
      ...

CALL OPEN (cust^file^name, cust^file^number,..);
  ...

cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop.
    CALL READ (cust^file^number, cust, cust^len);
    IF > THEN cust^eof := 1
    ELSE
      IF < THEN ... ! error.
      ELSE
        BEGIN ! process the record.
         ...
        END;
  END;  ! read loop.
   ...
```

Primary Key ↓

1	ADAMS	MIAMI, FL.	SO	0000.00	0500.00
2	BROWN,A	REEDLEY, CA.	WE	0256.95	0300.00
3	BROWN,B	BOSTON, MA.	EA	0301.00	1000.00
4	EVANS	BUTTE, MT.	WE	0010.00	0100.00
5	HARTLEY	CHICAGO, IL.	NO	0433.29	0500.00
6	JONES	DALLAS, TX.	SO	1234.56	2000.00
7	KOTTER	NEW YORK, NY.	EA	0089.00	0500.00
8	SMITH	DAYTON, OH.	NO	0010.00	0500.00
9	ROGERS	BOISE, ID.	WE	1024.00	1500.00
10	SANFORD	L.A., CA.	WE	0301.00	1000.00
11	SMITH	DAYTON, OH.	NO	0010.00	0500.00
12	EOF				

Example 3. Approximate Subset by Alternate Key

Key specifier = "RG".

```
CALL KEYPOSITION ( cust^file^number, key, "RG" );
                                        ! position to first record.
cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop.
    CALL READ (cust^file^number, cust, cust^len);
    IF > THEN cust^eof := 1
    ELSE
      IF < THEN ... ! error.
      ELSE
        BEGIN ! process the record.
         ...
        END;
  END;  ! read loop.
   ...
```

Alternate Key "RG"

1	BROWN,B	BOSTON, MA.	EA	0301.00	1000.00
2	KOTTER	NEW YORK, NY.	EA	0089.00	0500.00
3	HARTLEY	CHICAGO, IL.	NO	0433.29	0500.00
4	RICHARDS	MINNI, MN.	NO	0000.00	0500.00
5	SMITH	DAYTON, OH.	NO	0010.00	0500.00
6	ADAMS	MIAMI, FL.	SO	0000.00	0500.00
7	JONES	DALLAS, TX.	SO	1234.56	2000.00
8	BROWN	REEDLEY, CA.	WE	0256.95	0300.00
9	EVANS	BUTTE, MT.	WE	0010.00	0100.00
10	ROGERS	BOISE, ID.	WE	1024.00	1500.00
11	SANFORD	L.A., CA.	WE	0301.00	1000.00
12	EOF				

Example 4. Generic Subset by Primary Key

Primary-key value = "BROWN".

```
key ':=' "BROWN";
compare^len := 5;
CALL KEYPOSITION
              ( cust^file^number, key, ,compare^len ,generic );
cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop.
    CALL READ (cust^file^number, cust, cust^len);
    IF > THEN cust^eof := 1 ! end-of-file.
    ELSE
      IF < THEN ... ! error.
      ELSE
        BEGIN ! process the record.
         ...
        END;
  END;  ! read loop.
  ...
```

1	BROWN,A	REEDLEY, CA.	WE	0256.95	0300.00
2	BROWN,B	BOSTON, MA.	EA	0301.00	1000.00
3	EOF				

Example 5. Exact Subset By Primary Key

```
key := " ";                                  ! blank the
key[1] ':=' key FOR cust^name^len - 1; ! key.

key ':=' "SMITH";
CALL KEYPOSITION ( cust^file^number, key,,, exact );

cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop.
    CALL READ (cust^file^number, cust, cust^len);
    IF > THEN cust^eof := 1 ! end-of-file.
    ELSE
      IF < THEN ... ! error.
      ELSE
        BEGIN ! process the record.
         ...
        END;
  END;  ! read loop.
  ...
```

```
1    | SMITH      | DAYTON, OH.     | NO | 0010.00 | 0500.00 |

2    EOF
```

Example 6. Exact Subset by Non-Unique Alternate Key

```
key ':=' "NO";
CALL KEYPOSITION ( cust^file^number, key, "RG",, exact );

cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop.
    CALL READ (cust^file^number, cust, cust^len);
    IF > THEN cust^eof := 1 ! end-of-file.
    ELSE
      IF < THEN ... ! error.
      ELSE
        BEGIN ! process the record.
         ...
        END;
  END;  ! read loop.
  ...
```

1	HARTLEY	CHICAGO, IL.	NO	0433.29	0500.00
2	RICHARDS	MINNI, MN.	NO	0000.00	0500.00
3	SMITH	DAYTON, OH.	NO	0010.00	0500.00
4	EOF				

Example 7. Insertion of a Record into a Key-Sequenced File

Record to be inserted =

HEATHCLIFF	PORTLAND, OR.	WE	0000.00	0500.00

```
cust := "   ";                                    ! Blank the customer
cust[1] ':=' cust FOR (cust^len + 1) / 2; ! record.

cust^name    ':=' "HEATHCLIFF";
cust^address ':=' "PORTLAND, OR.";
cust^region  ':=' "WE";
cust^curbal   :=  0.00F;
cust^limit    :=  500.00F;
 ...
CALL WRITE (cust^file^number, cust, cust^len); ! insert a
                                               ! new record.
IF <> THEN ... ! error.
 ...
```

"CUSTOMER" File after Insertion

name	address	region	curbal	limit	
ADAMS	MIAMI, FL.	SO	0000.00	0500.00	
BROWN,A	REEDLEY, CA.	WE	0256.95	0300.00	
BROWN,B	BOSTON, MA.	EA	0301.00	1000.00	
EVANS	BUTTE, MT.	WE	0010.00	0100.00	
HARTLEY	CHICAGO, IL.	NO	0433.29	0500.00	
HEATHCLIFF	PORTLAND, OR.	WE	0000.00	0500.00	<-in-
JONES	DALLAS, TX.	SO	1234.56	2000.00	serted
KOTTER	NEW YORK, NY.	EA	0089.00	0500.00	
RICHARDS	MINNI, MN.	NO	0000.00	0500.00	
ROGERS	BOISE, ID.	WE	1024.00	1500.00	
SANFORD	L.A., CA.	WE	0301.00	1000.00	
SMITH	DAYTON, OH.	NO	0010.00	0500.00	

Example 8. Random Update

```
| EVANS    | BUTTE, MT.     | WE | 0010.00 | 0100.00 |
| HARTLEY  | CHICAGO, IL.   | NO | 0433.29 | 0500.00 | <-
| JONES    | DALLAS, TX.    | SO | 1234.56 | 2000.00 |

key := " ";                                  ! blank the
key[1] ':=' key FOR cust^name^len - 1; ! key.

key ':=' "HARTLEY";
CALL KEYPOSITION (cust^file^number, key);
IF <> THEN ...
CALL READUPDATE (cust^file^number, cust, cust^len);
IF <> THEN ...

   | HARTLEY    | CHICAGO, IL.   | NO | 0433.29 | 0500.00 |
 ...
cust^curbal := cust^curbal + 30.00F
 ...
CALL WRITEUPDATE (cust^file^number, cust, cust^len);
IF <> THEN ...

   | HARTLEY    | CHICAGO, IL.   | NO | 0463.29 | 0500.00 |
```

Example 9. Random Update to Non-Existent Record

Record to be updated = "BROWN, C"

ADAMS	MIAMI, FL.	SO	0000.00	0500.00
BROWN,A	REEDLEY, CA.	WE	0256.95	0300.00
BROWN,B	BOSTON, MA.	EA	0301.00	1000.00
EVANS	BUTTE, MT.	WE	0010.00	0100.00

```
key := " ";                                  ! blank the
key[1] ':=' key FOR cust^name^len - 1; ! key.

key ':=' "BROWN,C";
CALL KEYPOSITION (cust^file^number, key);
IF <> THEN ...
CALL READUPDATE (cust^file^number, cust, cust^len);
IF < THEN
  BEGIN
    CALL FILEINFO (cust^fum, error);
    IF error = 11 THEN .. ! record not found.
    ...
```

Example 10. Sequential Reading via Primary Key with Updating

The "limit" for each record having a "limit" >= 1000.00 and <= 2000.00 is raised to 2000.00.

```
compare^len := 0;
CALL KEYPOSITION ( cust^file^number, key, , compare^len);
                    ! position to first record via primary key.
cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop.
    CALL READ ( cust^file^number, cust, cust^len);
    IF > THEN cust^eof := 1
    ELSE
      IF < THEN ... ! error.
      ELSE
        BEGIN ! process the record.
          IF cust^limit >= 1000.00F
          AND cust^limit <= 2000.00F THEN
            BEGIN
              cust^limit := 2000.00F;
              CALL WRITEUPDATE
                              (cust^file^number, cust, cust^len);
              IF < THEN ... ! error.
              ...
            END;
        END;
  END;  ! read loop.
  ...
```

			limit		
ADAMS	MIAMI, FL.	SO	0000.00	0500.00	
BROWN,A	REEDLEY, CA.	WE	0256.95	0300.00	
BROWN,B	BOSTON, MA.	EA	0301.00	2000.00	<-in-
EVANS	BUTTE, MT.	WE	0010.00	0100.00	serted
HARTLEY	CHICAGO, IL.	NO	0463.29	0500.00	
JONES	DALLAS, TX.	SO	1234.56	2000.00	
KOTTER	NEW YORK, NY.	EA	0089.00	0500.00	
RICHARDS	MINNI, MN.	NO	0000.00	0500.00	
ROGERS	BOISE, ID.	WE	1024.00	2000.00	<---+
SANFORD	L.A., CA.	WE	0301.00	2000.00	<---+
SMITH	DAYTON, OH.	NO	0010.00	0500.00	|

inserted

Example 11. Random Deletion via Primary Key

Primary key = "EVANS"

BROWN,B	BOSTON, MA.	EA	0301.00	2000.00	
EVANS	BUTTE, MT.	WE	0010.00	0100.00	<-
HARTLEY	CHICAGO, IL.	NO	0463.29	0500.00	

```
key := " ";                                  ! blank the
key[1] ':=' key FOR cust^name^len - 1; ! key.

key ':=' "EVANS"
CALL KEYPOSITION (cust^file^number, key);
IF <> THEN ...
CALL WRITEUPDATE (cust^file^number, cust, 0);
IF <> THEN ...
```

BROWN,B	BOSTON, MA.	EA	0301.00	2000.00
HARTLEY	CHICAGO, IL.	NO	0463.29	0500.00

Example 12. Sequential Reading via Primary Key with Deletions

Each record having a "curbal" value of 0.00 is deleted.

```
CALL KEYPOSITION ( cust^file^number, key, , 0);    !position to
                                    ! first record via primary key.
cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop.
    CALL READ ( cust^file^number, cust, cust^len);
    IF > THEN cust^eof := 1
    ELSE
      IF < THEN ... ! error.
      ELSE
        BEGIN ! process the record.
          IF cust^curbal = 0.00F THEN
            BEGIN
              CALL WRITEUPDATE ( cust^file^number, cust, 0 );
              IF < THEN ... ! error.
              ...
            END;
        END;
  END;  ! read loop.
  ...
```

			curbal	
BROWN,A	REEDLEY, CA.	WE	0256.95	0300.00
BROWN,B	BOSTON, MA.	EA	0301.00	2000.00
HARTLEY	CHICAGO, IL.	NO	0463.29	0500.00
JONES	DALLAS, TX.	SO	1234.56	2000.00
KOTTER	NEW YORK, NY.	EA	0089.00	0500.00
ROGERS	BOISE, ID.	WE	1024.00	2000.00
SANFORD	L.A., CA.	WE	0301.00	2000.00
SMITH	DAYTON, OH.	NO	0010.00	0500.00

Example 13. Positioning with a Relative or Entry-Sequenced File

For these declarations:

```
INT(32) rec^addr;
STRING primary^key = rec^addr;
```

positioning by primary key is done with:

```
CALL POSITION (file^number, rec^addr);
```

positioning to end-of-file is done with:

```
rec^addr := -1D;
CALL POSITION (file^number, -1D);
```

and the current primary-key value (current position) can be obtained with either:

```
CALL FILEINFO (file^number,,,,,,,,,,,rec^addr);
   or
CALL FILERECINFO (file^number,,,,primary^key);
```

Example 14. Sequential Reading of a Relative or Entry-Sequenced File

Reading begins at the beginning of the file.

```
 ...
CALL OPEN (file^name, file^number,..);
 ...

eof := 0;
WHILE NOT eof DO
  BEGIN ! read loop.
    CALL READ (file^number, buffer, len, numread);
    IF > THEN eof := 1
    ELSE
      IF < THEN ... ! error.
      ELSE
        BEGIN ! process the record.
         ...
        END;
  END;  ! read loop.
  ...
```

Note that the preceding statements are functionally identical to the example for sequential access of a key-sequenced file via its primary key.

Example 15. Insertion to a Specific Position in a Relative File

```
CALL POSITION (file^number, 12345D);
CALL WRITE (file^number, buffer, count);
IF < THEN
  BEGIN
    CALL FILEINFO (file^number, error);
    IF error = 10 THEN ... ! record already exists at 12345D.
    ...
  END;
```

Example 16. Appending to the End of a Relative File

```
CALL POSITION (file^number, -1D);
WHILE 1 DO
  BEGIN
    ...
    buffer ':=' data FOR (count + 1)/2;

         prepare a record to be written.
    ...
    CALL WRITE (file^number, buffer, count);
    IF <> THEN   ... ! error.
    ...
    CALL FILERECINFO (file^number,,,,primary^key);

       returns the <record number> of where the new record is
       appended.

    ...
  END;
```

Example 17. Insertion to Empty Positions in a Relative File

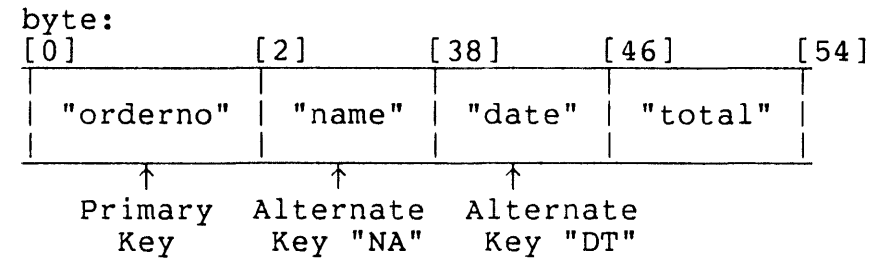
```
CALL POSITION (file^number, -2D);
WHILE 1 DO
  BEGIN
    ...
    buffer ':=' data FOR (count + 1)/2;
    ...
    CALL WRITE (file^number, buffer, count);
    IF <> THEN   ... ! error.
    ...
    CALL FILERECINFO (file^number,,,,primary^key);

       returns the <record number> of where the new record is
       appended.

    ...
  END;
```

Example 18. Appending Records to an Entry-Sequenced File

```
WHILE 1 DO
  BEGIN
    ...
    buffer ':=' data FOR (count + 1)/2;
    ...
    CALL WRITE (file^number, buffer, count);
    IF <> THEN   ... ! error.

    ...
  END;
```

RELATIONAL PROCESSING EXAMPLE

An "ORDER" Record:

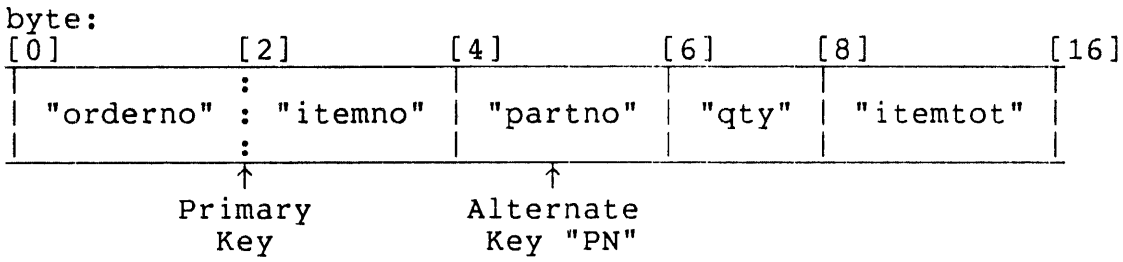



```
INT .order[0:26];                          ! order record.

STRING
    .sorder := @order '<<' 1;              ! byte addressable.

INT .order^orderno  = order;               ! order number field.

DEFINE
     order^len          =  54#,            ! order record length.
     order^name         =  sorder[2]#,     ! name field.
     order^name^len     =  36#,            ! name field length.
     order^date         =  sorderhrd[38]#, ! date field.
     order^date^len     =  8#;             ! date field length.

FIXED(2)
    .order^total        := @order[23];     ! total field.
                                           ! "total" = 0 means
                                           ! order not filled.
                                           ! "total" <> 0 means
                                           ! order filled but not
                                           ! shipped.
```

Contents of the "ORDER" File:

orderno	name	date	total
0020	SMITH	76/09/30	0000.00
0021	JONES	76/10/01	0000.00
0176	BROWN,B	76/10/17	0000.00
0180	ADAMS	76/10/17	0000.00
0410	SANFORD	76/10/22	0000.00
0498	ROGERS	76/11/02	0000.00
0568	EVANS	76/11/05	0000.00
0601	SMITH	76/11/08	0000.00
0621	RICHARDS	76/11/12	0000.00
0622	HARTLEY	76/11/12	0000.00
0623	KOTTER	76/11/12	0000.00

An "ORDER DETAIL" Record:

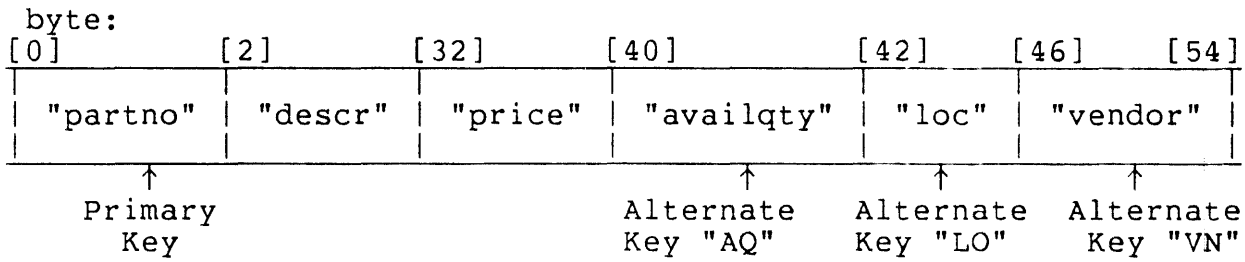



```
INT .orderdet[0:7];                          ! order detail record.

INT(32)
    .orderdet^orditem   := @orderdet;        ! order-item field.

DEFINE
     orderdet^len       =  16#,              ! order record length.
     orderdet^orderno   =  orderdet#,        ! order num. subfield.
     orderdet^itemno    =  orderdet[1]#,     ! item number subfield.
     orderdet^partno    =  orderdet[2]#,     ! part number field.
     orderdet^qty       =  orderdet[3]#;     ! quantity field.

FIXED(2)
    .orderdet^itemtot   := @orderdet[4];     ! item total field.
                                             ! total = 0 means item
                                             ! not available.
```

Contents of the "ORDER DETAIL" File:

orderno:itemno	partno	qty	itemtot
0020 : 0001	23167	00002	0000.00
0020 : 0002	02010	00001	0000.00
0020 : 0003	12950	00005	0000.00
0021 : 0001	00512	00022	0000.00
0021 : 0002	23167	00001	0000.00
0176 : 0001	32767	00001	0000.00
0180 : 0001	12950	00005	0000.00
0180 : 0002	32767	00022	0000.00
0180 : 0003	23167	00002	0000.00
0410 : 0001	01234	00010	0000.00
0410 : 0002	03379	00010	0000.00
...			
0623 : 0012	01234	00010	0000.00

An "INVENTORY" Record:




```
INT .inv[0:27];                          ! inventory record.

STRING
    .sinv := @inv '<<' 1;                ! byte addressable.

DEFINE
     inv^len              = 54#,         ! inventory record len.
     inv^partno           = inv#,        ! part number field.
     inv^descr            = sinv[2]#,    ! part descript. field.
     inv^descr^len        = 30#,         ! descr. field length.
     inv^availqty         = inv[20]#,    ! avail. quant. field.
     inv^location         = sinv[42]#,   ! location field.
     inv^location^len     = 4#,          ! loc. field length.
     inv^vendor           = sinv[46]#,   ! vendor field.
     inv^vendor^len       = 8#;          ! vendor field length.

FIXED(2)
    .inv^price := @inv[16];              ! price field.
```

Contents of "INVENTORY" file:

partno	descr	price	availqty	loc	vendor
00002	HI-FI	0129.95	00050	A01	TAYLOR
00512	RADIO	0010.98	00022	G10	GRAND
00987	TV SET	0200.00	00122	A76	TAYLOR
02010	TOASTER	0022.50	00000	F22	ACME
03379	CLOCK	0011.75	00512	A32	ZARF
12950	TOASTER	0020.45	00010	C98	SMYTHE
20211	WASHER	0314.29	00005	B44	SOAPY
		...			
23167	ANTENNA	0022.50	00008	A01	TAYLOR
32767	IRON	0025.95	00051	A82	HOT
65535	DRYER	0299.50	00022	Z02	SOAPY

The next example finds orders more than one month old and fills them. This involves these steps:

1. Read the "order" file sequentially via the date field.

2. When an order is found that must be filled, the corresponding "customer" record is read (random processing) and an order header consisting of customer name and address is printed.

3 Next, the generic subset in the "order detail" file corresponding to the current "orderno" is read sequentially.

4. For each line item (a record in the generic subset), the "inventory" file is read and updated (random processing), the line item record is updated, and the line item is printed.

5. When all line items for the current order have been processed, the order record is updated with the total price of the order. Then the customer current balance is updated and the total are printed.

The example code is

```
compare^len := 0;
! position to beginning of file via date field.
CALL KEYPOSITION (order^file^number,key,"DT",compare^len);
order^eof := 0;
WHILE NOT order^eof DO
  BEGIN ! reading order file via date field.
    CALL READ (order^fum, order, order^len);
    IF > OR order^date >= limit^date THEN order^eof := 1
    ELSE
      BEGIN ! fill order.
        ! read customer file.
        CALL KEYPOSITION (cust^file^number, order^name);
        CALL READUPDATE (cust^file^number, cust, cust^len);
        PRINT (order header);
        ! read order detail file for current order.
        compare^len := 2;
        CALL KEYPOSITION (orderdet^file^number,order^orderno,,
                          compare^len, generic);
        orderdet^eof := 0;
        WHILE NOT orderdet^eof DO
          BEGIN
            ! read line item.
            CALL READ
                  (orderdet^file^number,orderdet,orderdet^len);
            IF > THEN orderdet^eof := 1
            ELSE
              BEGIN
                CALL KEYPOSITION
                              (inv^file^number,orderdet^partno);
                CALL READUPDATE (inv^file^number,inv,inv^len);
                ! next, update inventory record
                CALL WRITEUPDATE (inv^file^number,inv,inv^len);
                ! next, update the order-detail record
                CALL WRITEUPDATE
                    (orderdet^file^number, orderdet, ordetlen);
                ! print the line item
                PRINT (line item)
              END;
          END;
        ! update the order file
        CALL WRITEUPDATE (order^file^number, order, order^len);
        ! update the customer file.
        CALL WRITEUPDATE (cust^file^number, cust, cust^len);
        PRINT (total);
      END; ! of fill order.
  END; ! of read order file via date field.
```

Records and files used to fill the first order:

From the "ORDER" file -

0020	SMITH	76/09/30	0000.00

From the "CUSTOMER" file -

SMITH	DAYTON, OH.	NO	0010.00	0500.00

From the "ORDER DETAIL" file -

0020 : 0001	23167	00002	0000.00
0020 : 0002	02010	00001	0000.00
0020 : 0003	12950	00005	0000.00

From the "INVENTORY" file -

23167	ANTENNA	0022.50	00008	A01	TAYLOR
02010	TOASTER	0022.50	00000	F22	ACME
12950	TOASTER	0020.45	00010	C98	SMYTHE

Records and files after filling the first order:

From the "ORDER" file -

0020	SMITH	76/09/30	0147.25

From the "CUSTOMER" file -

SMITH	DAYTON, OH.	NO	0157.25	0500.00

From the "ORDER DETAIL" file -

0020 : 0001	23167	00000	0045.00	
0020 : 0002	02010	00001	0000.00	<- not filled
0020 : 0003	12950	00000	0102.25	

From the "INVENTORY" file -

23167	ANTENNA	0022.50	00006	A01	TAYLOR	
02010	TOASTER	0022.50	00000	F22	ACME	<-none
12950	TOASTER	0020.45	00005	C98	SMYTHE	

SECTION 6

FILE LOADING

The File Utility Program (FUP) commands related to loading data into an existing file are LOAD, LOADALTFILE, and BUILDKEYRECORDS.

- The LOAD command loads data into an existing structured disc file without affecting any alternate-key files. Existing data in the file being loaded are lost.

  For loading of key-sequenced files, the input records can be in sorted or unsorted order (unsorted is assumed unless the SORTED option is specified). Also for key-sequenced files, the slack space (the percentage of data block and index block space to be left for future insertions) can be specified.

- The LOADALTFILE command generates alternate-key records from a specified primary-key file and loads those records into the specified alternate-key file. Slack space for future insertions can be specified. LOADALTFILE always sorts the alternate-key records before actually loading them.

- Because some systems may have insufficient disc space for the sorting operation, the alternate-key file can be loaded in two separate steps by first performing a BUILDKEYRECORDS operation.

  The BUILDKEYRECORDS command generates alternate-key records for specified key fields of a specified structured disc file and writes those records to a designated file (not necessarily the ultimate-destination alternate-key file). This output (which can be to a magnetic-tape file) can then be loaded into the alternate-key file by means of a COPY or LOAD command.

FUP resides in volume $SYSTEM. Normally, it is run through use of the GUARDIAN Operating System Command Interpreter. For a complete description of all the FUP commands, see the GUARDIAN Operating System Utilities Reference Manual. Also see the FUP discussion in the GUARDIAN Operating System User's Guide.

The examples below illustrate file-loading operations that require a sequence of FUP commands. Example 1 loads a key-sequenced file. Example 2 adds an alternate key to a file having alternate keys. Example 3 adds an alternate key to a file not having alternate keys. Example 4 reloads a single partition of a partitioned key-sequenced file. Example 5 loads a single partition of a partitioned alternate-key file.

EXAMPLE 1: LOAD A KEY-SEQUENCED FILE

For this example, file $VOL1.SVOL.PARTFILE is a key-sequenced file having three partitions. The first secondary partition is $VOL2 and the second secondary partition is $VOL3.

Any record having a primary-key value in the range of zero to (but not including) HA are to exist in the primary partition; records with primary-key values from HA to (but not including) RA are to exist on $VOL2; records with primary-key values of RA or greater are to exist on $VOL3.

The records to be loaded into this file are 128 bytes long and are on tape in unsorted order. The tape is written with one record per block.

The FUP commands to perform this operation are

```
-VOLUME $vol1.subvolume
-LOAD $TAPE, partfile
```

LOAD reads the records from tape drive $TAPE and sends them to the SORT process. When all records have been read, sorting begins. When the sort is finished, the records are read from the SORT process and loaded into the file according to the file's <partial-key value> specifications. The data and index block slack percentage is zero (0).

EXAMPLE 2: ADD AN ALTERNATE KEY TO A FILE HAVING AN ALTERNATE KEY

This example adds an alternate key to primary file $VOL1.SVOL.PRIFILE, which has one alternate-key file designated $VOL1.SVOL.ALTFILE. The alternate-key records for the new key field will be added to file ALTFILE.

The <key specifier> for the new key is NM, the <key offset> in the record is four (4), the <key length> is twenty (20), a "null value" of " " (blank) is specified for the new key field.

The FUP commands to perform this operation are

```
-VOLUME $vol1.subvolume
-ALTER prifile, ALTKEY ( "NM", KEYOFF 4, KEYLEN 20, NULL " " )
-LOADALTFILE 0, prifile, ISLACK 10
```

The LOADALTFILE command loads PRIFILE's key file zero (0), $VOL1.SVOL.ALTFILE, with the alternate-key records for <key specifier> NM and for any other alternate keys defined for key file zero (0). An index-block slack percentage of 10 is specified.

EXAMPLE 3: ADD AN ALTERNATE KEY TO A FILE NOT HAVING ALTERNATE KEYS

This example adds an alternate key to primary file $VOL1.SVOL.FILEA, which is an entry-sequenced file. The new alternate-key file will be $VOL1.SVOL.FILEB. The alternate-key records for the new key field will be added to FILEB.

The <key specifier> for the new key is XY, the <key offset> in the record is (0), and the <key length> is ten (10).

The FUP commands to perform this operation are

```
-VOLUME $vol1.subvolume
-CREATE fileb, type K, rec 16, keylen 16
-ALTER filea, ALTFILE ( 0, fileb ), ALTKEY ( "XY", KEYLEN 10 )
-LOADALTFILE 0, filea
```

The CREATE command creates the alternate-key file $VOL1.SVOL.FILEB. The record length and key length are specified as 16 bytes (2 for the key specifier + 10 for the alternate-key-field lengths + 4 for the primary-key length).

The ALTER command changes the file label for FILEA so that it refers to FILEB as <alternate-key file> 0, and contains the definition for the key field specified by <key specifier> XY.

The LOADALTFILE command loads FILEA's key file zero (0), $VOL1.SVOL.FILEB, with the alternate-key records for <key specifier> XY. An index-block-slack percentage of zero (0) is implied.

EXAMPLE 4: RELOAD A SINGLE PARTITION OF A PARTITIONED, KEY-SEQUENCED FILE

For this example, the primary partition of the partitioned file is $VOL1.SVOL.PARTFILE. Its first secondary partition is on $VOL2 and its second secondary partition is on $VOL3. The secondary partition on $VOL2 is to be loaded.

The FUP commands to perform this operation are

```
-VOLUME $vol1.subvolume
-SET LIKE $vol2.partfile
-SET NO PARTONLY
-CREATE temp
-DUP $vol2.partfile, temp, OLD, PARTONLY
-LOAD temp, $vol2.partfile, SORTED, PARTOF $vol1
-PURGE temp
```

The SET and CREATE commands create a file identical to $VOL2.SVOL.PARTFILE except that the file is designated a non-partitioned file by means of NO PARTONLY.

The DUP command duplicates the data in the secondary partition ($VOL2.SVOL.PARTFILE) into $VOL1.SVOL.TEMP.

The LOAD command reloads the secondary partition $VOL2.SVOL.PARTFILE. The SORTED option is specified because the records in the TEMP file are already in sorted order.

EXAMPLE 5: LOAD A SINGLE PARTITION OF A PARTITIONED, ALTERNATE-KEY FILE

For this example, primary file $VOL1.SVOL.PRIFILE is a key-sequenced file having a primary-key field ten bytes long. It has three alternate-key fields defined by the <key specifiers> F1, F2, and F3. Each of these alternate-key fields is ten bytes long.

All alternate-key records are contained in one alternate-key file that is partitioned over three volumes; each volume contains the alternate-key records for one alternate-key field (the <key specifier> for the alternate-key field is also the <partial-key value> for the secondary partitions).

The alternate-key file's primary partition is $VOL1.SVOL.ALTFILE. It contains the alternate-key records for the <key specifier> F1. The first secondary partition, $VOL2.SVOL.ALTFILE, contains the alternate-key records for <key specifier> F2. The second secondary partition, $VOL3.SVOL.ALTFILE, contains the alternate-key records for <key specifier> F3.

To load the alternate-key records for <key specifier> F2 into $VOL2.SVOL.ALTFILE, the FUP commands are:

```
-VOLUME $vol1.subvolume
-CREATE sortin, ext 30
-CREATE sortout, ext 30
-BUILDKEYRECORDS prifile,sortin,"F2",RECOUT 22,BLOCKOUT 2200
-EXIT
:SORT
<FROM sortin, RECORD 22
<TO sortout
<ASC 1:22
<RUN
<EXIT
:FUP
-VOLUME $vol1.subvolume
-LOAD sortout, $vol2.altfile, SORTED, PARTOF $vol1, RECIN 22,
   BLOCKIN 2200
-PURGE ! sortin, sortout
```

The CREATE commands create the disc file used as the output of BUILDKEYRECORDS (which is also the input to SORT) and the disc file to be used as the output of SORT.

The BUILDKEYRECORDS command generates the alternate-key records for <key specifier> F2 of PRIFILE and writes the records to SORTIN. Record-blocking is used, to improve the efficiency of disc writes.

The SORT program sorts the alternate-key records. The key-field length for the sort is the same as the alternate-key record length (22, 2 for the <key specifier> + 10 for alternate-key-field length + 10 for the primary-key-field length). The output file of the sort is SORTOUT.

The LOAD command loads the secondary partition $VOL2.SVOL.ALTFILE with the alternate-key records for <key specifier> F2. Note that the record-blocking here is complementary to that used with BUILDKEYRECORDS.

APPENDIX A

ASCII CHARACTER SET

The table below shows the USA Standard Code for Information Interchange (ASCII) character set, and the corresponding code values in octal notation.

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
NUL	000000	000000	Null
SOH	000400	000001	Start of heading
STX	001000	000002	Start of text
ETX	001400	000003	End of text
EOT	002000	000004	End of transmission
ENQ	002400	000005	Enquiry
ACK	003000	000006	Acknowledge
BEL	003400	000007	Bell
BS	004000	000010	Backspace
HT	004400	000011	Horizontal tabulation
LF	005000	000012	Line feed
VT	005400	000013	Vertical tabulation
FF	006000	000014	Form feed
CR	006400	000015	Carriage return
SO	007000	000016	Shift out
SI	007400	000017	Shift in
DLE	010000	000020	Data link escape
DC1	010400	000021	Device control 1
DC2	011000	000022	Device control 2
DC3	011400	000023	Device control 3
DC4	012000	000024	Device control 4
NAK	012400	000025	Negative acknowledge
SYN	013000	000026	Synchronous idle
ETB	013400	000027	End of transmission block

→

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
CAN	014000	000030	Cancel
EM	014400	000031	End of medium
SUB	015000	000032	Substitute
ESC	015400	000033	Escape
FS	016000	000034	File separator
GS	016400	000035	Group separator
RS	017000	000036	Record separator
US	017400	000037	Unit separator
SP	020000	000040	Space
!	020400	000041	Exclamation point
"	021000	000042	Quotation mark
#	021400	000043	Number sign
$	022000	000044	Dollar sign
%	022400	000045	Percent sign
&	023000	000046	Ampersand
'	023400	000047	Apostrophe
(	024000	000050	Opening parenthesis
)	024400	000051	Closing parenthesis
*	025000	000052	Asterisk
+	025400	000053	Plus
,	026000	000054	Comma
-	026400	000055	Hyphen (minus)
.	027000	000056	Period (decimal point)
/	027400	000057	Right slant
0	030000	000060	Zero
1	030400	000061	One
2	031000	000062	Two
3	031400	000063	Three
4	032000	000064	Four
5	032400	000065	Five
6	033000	000066	Six
7	033400	000067	Seven
8	034000	000070	Eight
9	034400	000071	Nine
:	035000	000072	Colon
;	035400	000073	Semicolon
<	036000	000074	Less than
=	036400	000075	Equals
>	037000	000076	Greater than
?	037400	000077	Question mark

→

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
@	040000	000100	Commercial "at"
A	040400	000101	Uppercase A
B	041000	000102	Uppercase B
C	041400	000103	Uppercase C
D	042000	000104	Uppercase D
E	042400	000105	Uppercase E
F	043000	000106	Uppercase F
G	043400	000107	Uppercase G
H	044000	000110	Uppercase H
I	044400	000111	Uppercase I
J	045000	000112	Uppercase J
K	045400	000113	Uppercase K
L	046000	000114	Uppercase L
M	046400	000115	Uppercase M
N	047000	000116	Uppercase N
O	047400	000117	Uppercase O
P	050000	000120	Uppercase P
Q	050400	000121	Uppercase Q
R	051000	000122	Uppercase R
S	051400	000123	Uppercase S
T	052000	000124	Uppercase T
U	052400	000125	Uppercase U
V	053000	000126	Uppercase V
W	053400	000127	Uppercase W
X	054000	000130	Uppercase X
Y	054400	000131	Uppercase Y
Z	055000	000132	Uppercase Z
[	055400	000133	Left square bracket
\	056000	000134	Left slant
]	056400	000135	Right square bracket
^	057000	000136	Circumflex
_	057400	000137	Underscore
•	060000	000140	Grave accent
a	060400	000141	Lowercase a
b	061000	000142	Lowercase b
c	061400	000143	Lowercase c
d	062000	000144	Lowercase d
e	062400	000145	Lowercase e
f	063000	000146	Lowercase f
g	063400	000147	Lowercase g

→

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
h	064000	000150	Lowercase h
i	064400	000151	Lowercase i
j	065000	000152	Lowercase j
k	065400	000153	Lowercase k
l	066000	000154	Lowercase l
m	066400	000155	Lowercase m
n	067000	000156	Lowercase n
o	067400	000157	Lowercase o
p	070000	000160	Lowercase p
q	070400	000161	Lowercase q
r	071000	000162	Lowercase r
s	071400	000163	Lowercase s
t	072000	000164	Lowercase t
u	072400	000165	Lowercase u
v	073000	000166	Lowercase v
w	073400	000167	Lowercase w
x	074000	000170	Lowercase x
y	074400	000171	Lowercase y
z	075000	000172	Lowercase z
{	075400	000173	Opening brace
|	076000	000174	Vertical line
}	076400	000175	Closing brace
~	077000	000176	Tilde
DEL	077400	000177	Delete

APPENDIX B

BLOCK FORMATS OF STRUCTURED FILES

This appendix describes the block formats for key-sequenced, entry-sequenced, and relative files. A block in a structured file usually consists of a header, a record area, and a map of offsets pointing to the beginning of each record. For a relative file under the DP2 disc process, an array of record lengths replaces the offsets map.

The block format for NonStop 1+ systems and for NonStop systems with DP1 disc processes is shown in Figure B-1 and described in detail on the subsequent pages.

Under the DP2 disc process, a key-sequenced file begins with a bit-map block telling which data and index blocks are in use. The second block is the root (highest-level) index block for the file. The third block is either a second-level index block or the file's first data block. Under DP1, a key-sequenced file's first block is the root index block.

In a relative file under DP2, the first block is a bit-map block telling which data blocks contain at least one record. The second block is always the first data block. A DP1 relative file's first block is a data block.

In an entry-sequenced or unstructured file under DP1 or DP2, all blocks are data blocks.

Figure B-2 shows the general block format for NonStop systems with DP2 disc processes. DP2's five different block-header structures are shown in Figures B-3 through B-8.

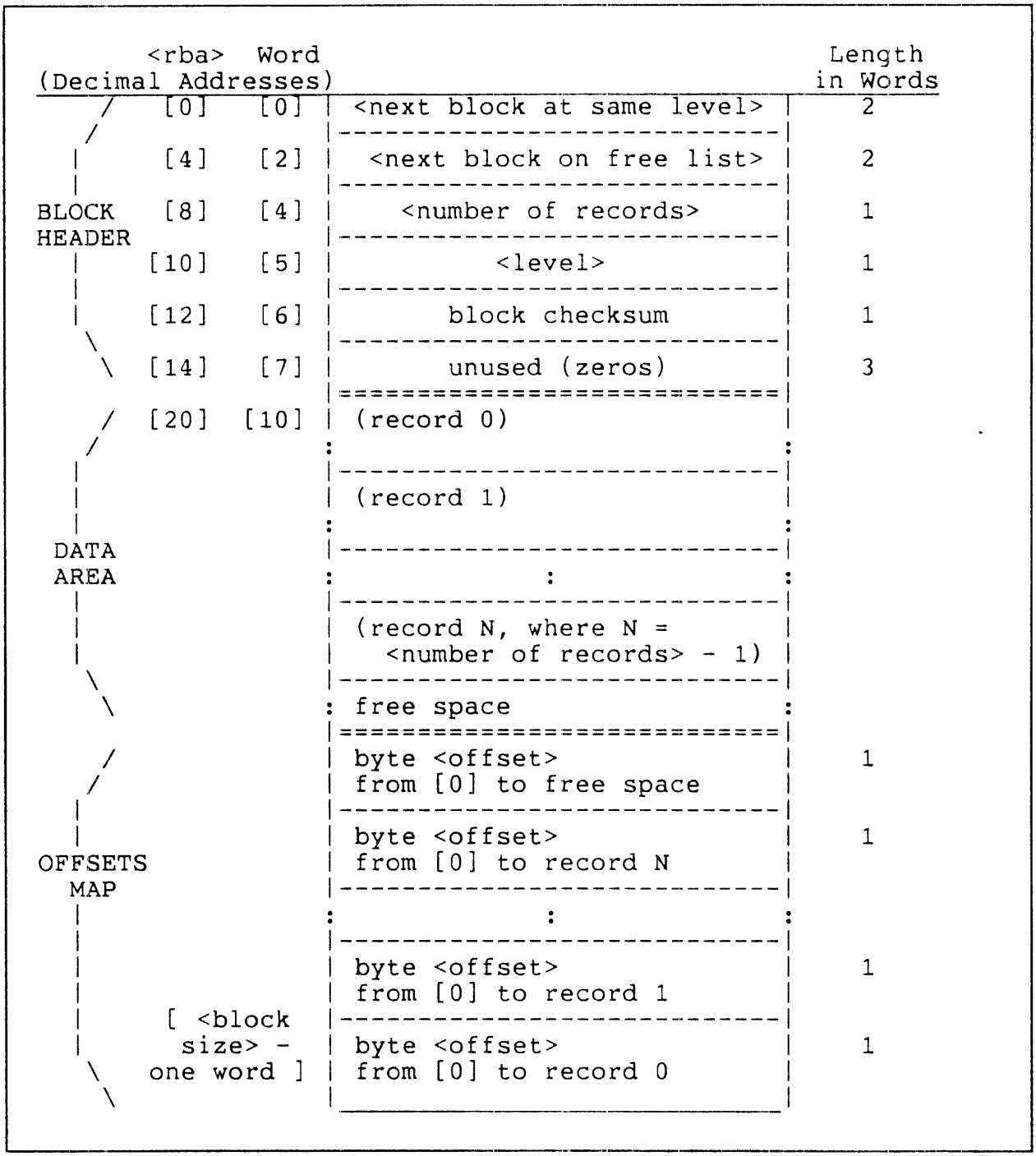


Figure B-1. Block Format for Structured Files (non-DP2 systems)

<rba>

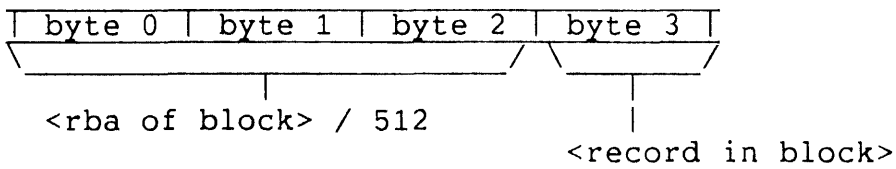
A block is addressed by a doubleword relative byte address (RBA). In a key-sequenced file, (<rba> = 0) points to the root (highest-level) index block; in a relative or entry-sequenced file, (<rba> = 0) points to the first data block.

To locate a given record in a key-sequenced file, the key value supplied to KEYPOSITION is used to search the block for a record having a key field that matches.

To locate a given record in a relative file, the <record number> supplied to POSITION is converted to a block address and record number in the block, as shown here:

```
<blocking factor> =
     ( <block length> - 22 ) / ( <create record length> + 2 )

<rba of block>    =
     <record number> / <blocking factor> * <block length>

<record in block> = <record number> '\' <blocking factor>
```

The format of a <record address> used to position to a record in an entry-sequenced file is

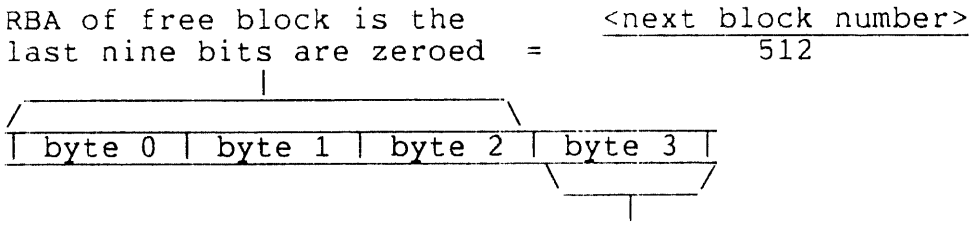


<next block at same level>

For key-sequenced files, this is the block number of the next block at the same level; for relative and entry-sequenced files, this field is not used and is set to zero.

<next block on free list>

For key-sequenced and relative files, this serves two functions: It holds (1) the block number of the next block on the file's free list(s) (for a key-sequenced file, if the index- and data-block lengths differ, the file has two free lists; one for index blocks and another one for data blocks) and (2) the number of free blocks on the list. The doubleword is formatted as shown here:



If the entry = -1D for a key-sequenced or relative file, the block is the last block in the free list.

If the entry = -2D for a key-sequenced file, the block is in use.

If the entry = -2D for a relative file, there are no free records in the block.

For entry-sequenced files, this entry is not used and is set to -1D.

<number of records>

is the number of records written in the block.

<level>

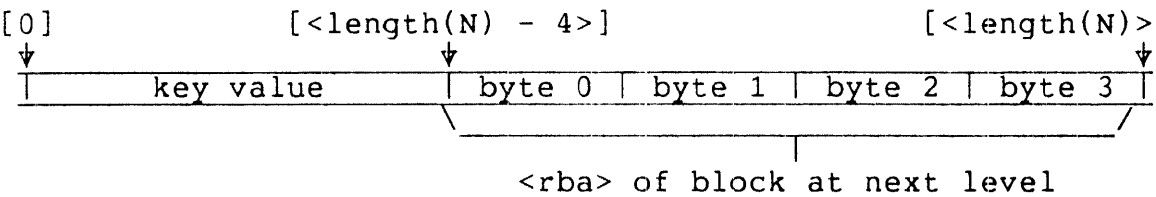
is the tree level of the block. <level> = 0 means the block is a data block; <level> > 0 means the block is an index block.

<record>

is a data record (if <level> = 0) or index record (if <level> > 0). The length of record N is

<offset to record N + 1> - <offset to record N>

Also, a record must be able to fit into the record area of one block. The format of a DP1 or NonStop 1+ index record is



<offset>

is the offset, in bytes, to the beginning of "free space" or a record. For relative files, the number of <offset>s is always the same as the <blocking factor>.

<block size>

is the block size in words, calculated as shown here:

<block size> = (<creation block length> + 1) / 2

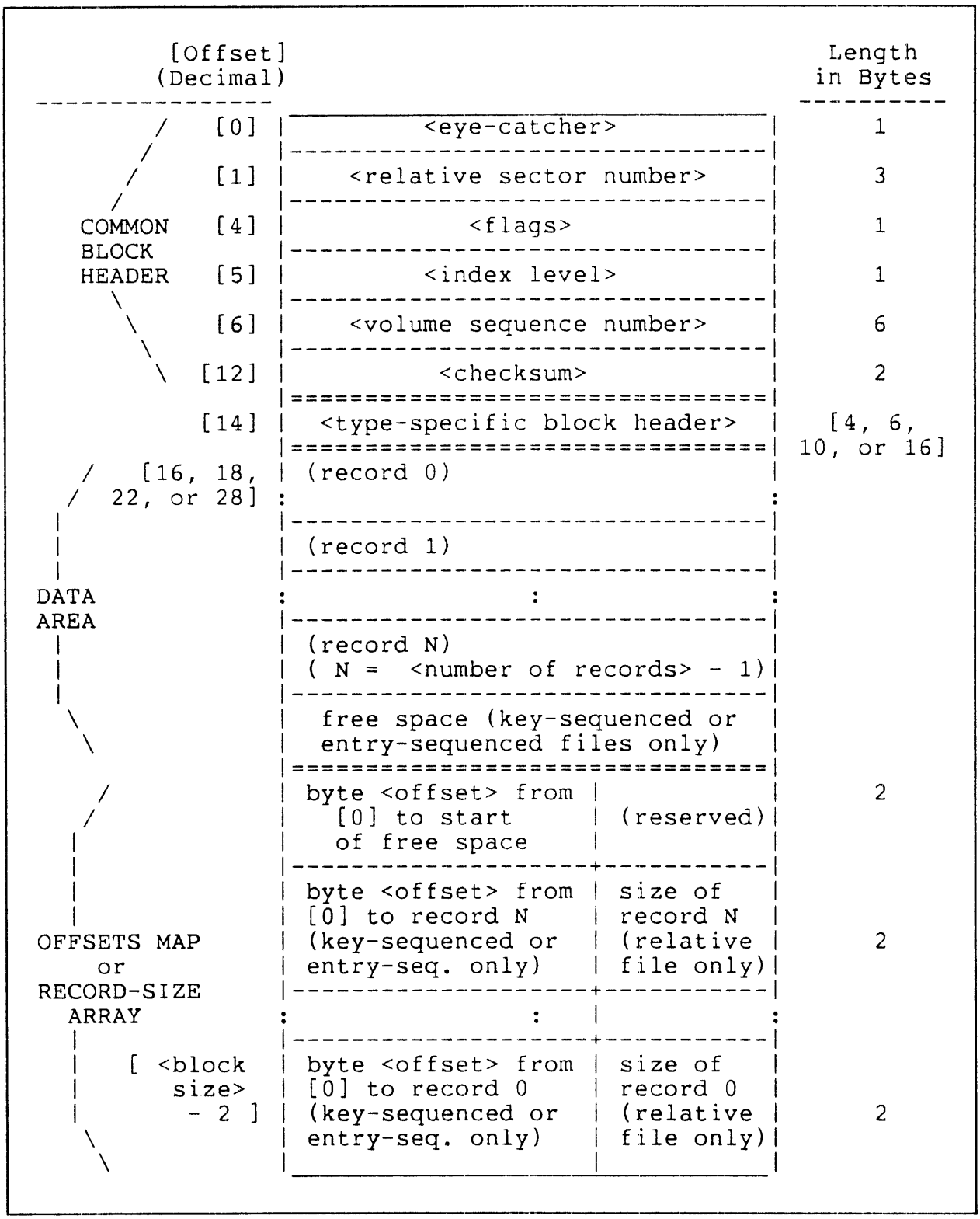


Figure B-2. Block Format for DP2 Structured Files

<eye catcher>

is currently set to ">" but may be changed in a future release.

<relative sector number>

identifies the relative 512-byte sector within the file.

<flags>

Bit 0: This bit is set (=1) if the block is broken (inconsistent).

Bits 3-5: These three bits indicate the file type as follows:

```
000 (reserved)
001 Relative File
010 Entry-Sequenced File
011 Key-Sequenced File
100 (reserved)
101 (reserved)
110 (reserved)
111 Directory
```

Bits 6-7: These two bits indicate the block type as follows:

```
00 Data or Index
01 Bit Map (file must be key-sequenced or relative)
10 Free (file must be key-sequenced)
11 (reserved)
```

<index level>

contains the tree level of the block. If the block is not an index block, <level> = 0.

<volume sequence number>

identifies the last update of a structured block. This number is incremented each time a change is made to the block, regardless of whether the block is written to disc. For a TMF-audited file, the <volume sequence number> is included in the audit-checkpoint (AC) record. Later, during autorollback or takeover, the number in the block header is compared with the number in the AC record to determine whether the AC record must be applied.

<checksum>

is the software checksum over the entire block.

<type-specific block header>

is the block-header area that differs according to the type of file. The various block headers are illustrated in Figures B-3 through B-7. Figure B-8 illustrates the arrangement of bit-map blocks with key-sequenced and relative files.

Offset (Decimal)		Length in Bytes
[0]	<common block header>	14
[14]	<quantity of records allocated>	2
[16]	(reserved)	8
[24]	(record 0)	

Figure B-3. Header for DP2 Key-Sequenced Index Block

Offset (Decimal)		Length in Bytes
[0]	<common block header>	14
[14]	<quantity of records allocated>	2
[16]	(reserved)	8
[24]	<relative sector number> of next data block	3
[27]	<relative sector number> of previous data block	3
[30]	(record 0)	

Figure B-4. Header for DP2 Key-Sequenced Data Block

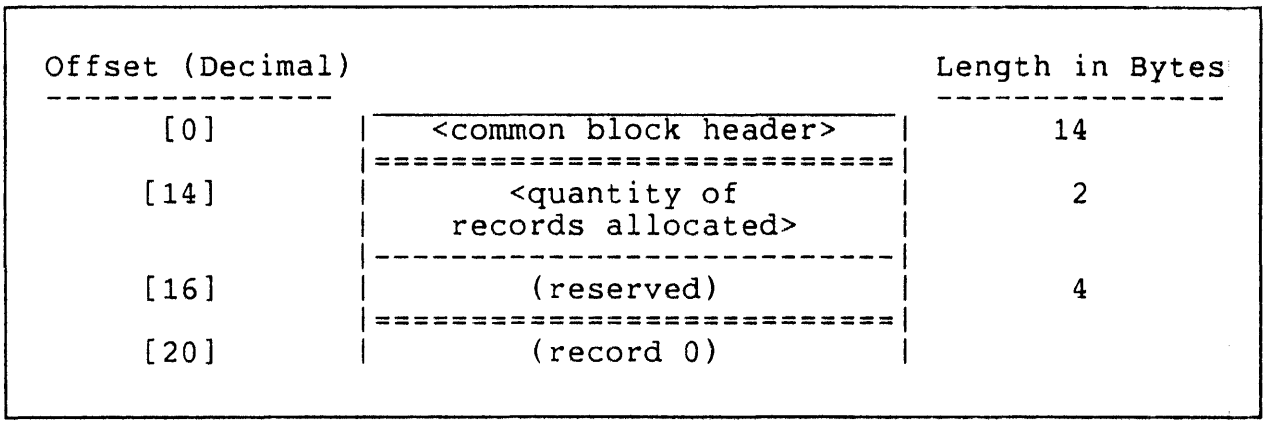


Figure B-5. Header for DP2 Entry-Sequenced Data Block

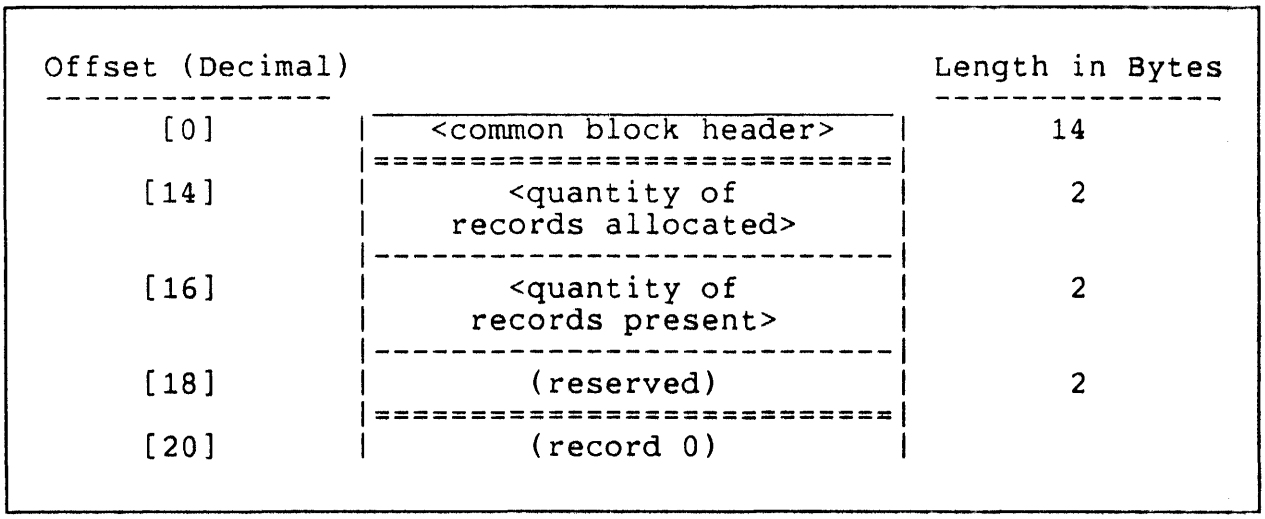


Figure B-6. Header for DP2 Relative Data Block

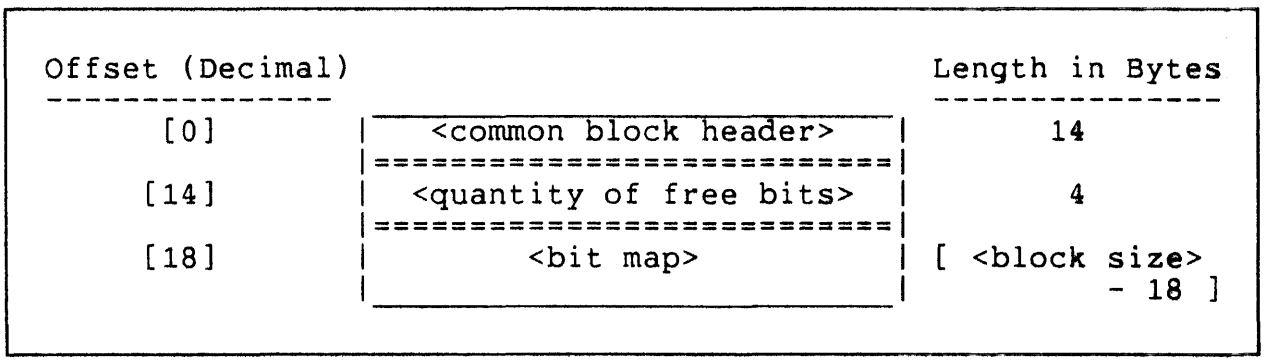


Figure B-7. Header for DP2 Bit-Map Block

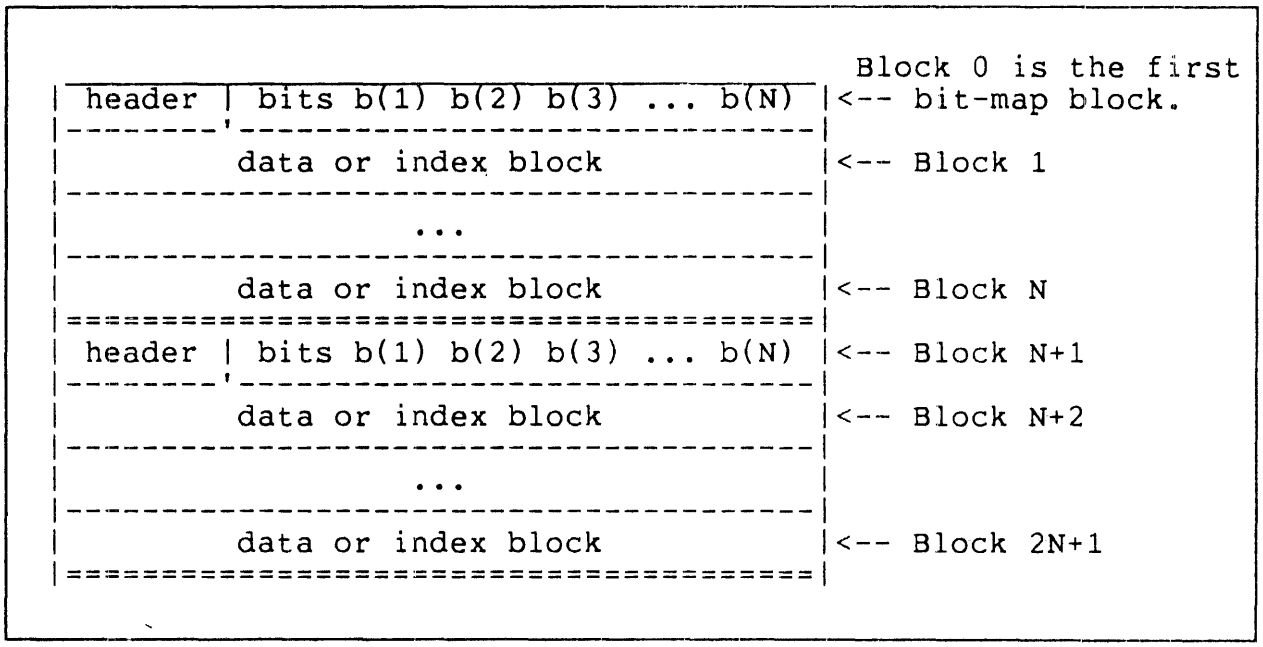


Figure B-8. Arrangement of DP2 Bit-Map Blocks

<common block header>

is illustrated in Figure B-2.

<quantity of records allocated>

tells how many records have been allocated in the block. This field occurs in the header of all four types of index or data blocks.

<quantity of records present>

tells how many records have been allocated in the block. This field occurs in the header of a relative data block only.

<relative sector number> of next data block

provides a link to the next logical block. The current block's <relative sector number> is given in the common block header. This field occurs in the header of a key-sequenced data block only.

<relative sector number> of previous data block

provides a link to the previous logical block. This field occurs in the header of a key-sequenced data block only.

<quantity of free bits>

tells how many bits in this bit-map indicate blocks which are free (empty) in a key-sequenced file or not full in a relative file. This field occurs in the header of a bit-map block only.

<bit map>

is an array of bits describing availability of index or data blocks. For a key-sequenced file, each bit tells whether the corresponding block is free (0) or in use (1). For a relative file, each bit tells whether there is room for at least one more record in the corresponding block. This is analogous to the "free lists" which enable space management under the DP1 disc process.

An empty bit-map has (8 * (<block size> - 18)) free bits. With a 1024-byte block, for example, the map has 8048 available bits.

<record>

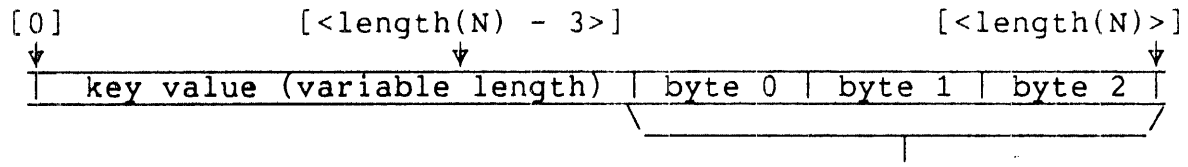
can be a data or index record. The length of record N in a key-sequenced or entry-sequenced file is

<offset to record N + 1> - <offset to record N>

A record must be able to fit into the record area of one block. Thus the maximum key-sequenced record length is the block size minus 34 (30 bytes for the header and 4 for the smallest possible offsets map).

DP2 index records differ from those of DP1 disc processes or NonStop 1+ systems because of the different internal structure of key-sequenced files. Where a DP1 index block uses a four-byte relative byte address as a pointer, DP2 uses a three-byte relative sector number. Where DP1 maintains a linked list of free blocks, DP2 uses bit-map blocks.

The format of a DP2 index record is:



The <relative sector number> field in an index record points to the start of the block associated with this key. A null key value is used when KEYLEN = 0; this occurs in any index record pointing to an index block or to the first data record.

<offset>

is the distance, in bytes, from the beginning of the block (offset = 0) to the beginning of a record or free space.

APPENDIX C

THE DP1 AND DP2 DISC PROCESSES

This appendix compares the major features of the DP1 and DP2 disc processes, then discusses file-system compatibility between DP1 volumes and DP2 volumes within the same system or network. Section 2 discusses the structures of various types of ENSCRIBE files.

COMPARISON OF DP1 AND DP2

Table C-1 shows how the DP1 and DP2 disc processes differ from each other.

Table C-1. Differences in DP1 and DP2 Disc Processes

Characteristic	DP1	DP2
Maximum record length (if block size = 4096)	Size-Header-Offsets	(see note #3) Size-Header-Offsets
key-sequenced	(4096-20-6)/2=2035*	4096-30-4=4062 bytes
relative	4096-20-4=4072 bytes	4096-20-4=4072 bytes
entry-sequenced	4096-20-4=4072 bytes	4096-20-4=4072 bytes
unstructured	4096 bytes	4096 bytes
	*DP1 requires that at least two records fit in a K-S block.	
Legal block sizes	512, 1024, 1536, 2048, 2560, 3072, 3584, or 4096 bytes	512, 1024, 2048, or 4096 bytes

Table C-1. Differences in DP1 and DP2 Disc Processes (continued)

Characteristic	DP1	DP2
Key-sequenced index and data blocks	Can be different sizes	Must be the same size
Block-header length		
key-seq. data	20 bytes	30 bytes
key-seq. index	20 bytes	24 bytes
relative	20 bytes	20 bytes
bit-map blocks	(do not exist)	18 bytes
entry-sequenced	20 bytes	20 bytes
Maximum number of records (N) in a block for different record lengths	N=(B-22)/(R+2) where B=Block length and R=Record length Also, N < 512	N=B-H(type)/(R+2) where H(KS)=30+2=32 H(REL)=20+2=22 H(ES)=20+2=22 (see Note 3)
Because the disc process uses a nine-bit, internal variable to store the record number, the maximum number of records of any size is 511.		
Maximum extents per file	16	As many as 978 (see Note 1); limit is dynamically extensible
Number of directory extents	1	As many as 987; the limit is dynamically extensible
Largest extent	65535 pages 134,215,680 bytes	(see Note 2)
Address Space (AS)		
key-sequenced	AS=physical storage	AS=physical storage minus bit-map block
relative	AS=physical storage	AS=physical storage minus bit-map block
entry-sequenced	AS=physical storage	AS=physical storage
unstructured	AS=physical storage	AS=physical storage

Table C-1. Differences in DP1 and DP2 Disc Processes (continued)

Characteristic	DP1	DP2
Cache	•Binary search •Not dynamic •Buffered for audited files; write-through for others •"Least-recently-used" (LRU) access mode	•Hashed search •Dynamic •Buffered for audited files; optional for others •Random (LRU), sequential, system-managed, or direct I/O
Maximum number of partitions	16	16
Maximum number of locks	In a NonStop system, 3000 concurrent record locks or 2000 key locks on a file. In a NonStop 1+ system, the limits are 1808 and 922, respectively.	Arbitrary limit of 2000 locks per user (that is, per opener or per transaction identifier)
Lock search	Sequential search on locks appended to an FCB	Hash search on lock table
Audited files and audit-trail files	May be on the same volume	May not be on the same volume
TMF monitor/master audit trail	Monitor audit trail contains commitment, and abortion records; separate audit trail contains data records	Master audit trail can contain commitment, abortion, and data records; auxiliary audit trail is optional

Note 1: The maximum number of extents per file can depend on the number of alternate keys, because extent and alternate-key information share the same area of the file label. For example, consider a structured file having two alternate keys:

Directory block size:	4096 bytes
Block header and trailer:	-34 bytes
Fixed part of DP2 file label:	-150 bytes
Information on two alternate keys:	-66 bytes
Space left for four-byte extent entries:	3846 bytes

The maximum number of extents allowable for this file is 3846/4 or 961 extents.

Note 2: DP2 extent allocation for relative and key-sequenced files includes bit-map blocks that are not accessible to the user for data storage. Therefore, a DP2 extent can hold slightly fewer data bytes than a DP1 extent of the same size.

Note 3: If there are N records in a block, then there are N+1 two-byte offset fields at the end of a key-sequenced or entry-sequenced block--one pointing to each record and one pointing to the free space. These offset fields and the header are not available for data storage, so they must be subtracted from the block size when you calculate available space.

If a relative file under DP2 has N records, then there are only N overhead words (each two bytes long) at the end of the block. One two-byte field is unused in a DP2 relative file, however, so a DP2 block can always be mapped back to a DP1 format to comply with a file system call from a DP1 node.

FILE-SYSTEM COMPATIBILITY BETWEEN DP1 AND DP2

The DP1 and DP2 disc processes treat files differently in many respects. This can cause some difficulties in operations dealing with mixtures of DP1 and DP2 volumes. These considerations concern mixed sets of volumes within a single system or among two or more systems in a network.

Detection of Version Levels

When a file is opened on an A03 or later NonStop system, the file system and the disc process exchange information about their respective version levels. NonStop 1+ systems neither send nor look at this version information.

Some interprocess messages differ with the version information at hand. For example, if a DP2 disc process is requested to perform

sequential block-buffering, it returns a block in DP2 format to a B00 (or later) file system and in DP1 format to any other file system.

Also, when a file is opened under certain conditions, an A06 (or later) disc process sends file-system error 60 to pre-A06 or NonStop 1+ file systems, but sends sequential I/O (SIO) error 59 to A06 or later file systems under the same conditions.

File Creation

Two forms of file-creation messages exist, one for DP1 and another for DP2. A B00 or later NonStop system determines which type of disc process is involved and sends the correct form of creation message. Other file systems send only the DP1 form. A DP2 disc process can receive and handle either form.

E08 and later NonStop 1+ systems accept, but do not use, the special creation parameters designed for DP2 files. These parameters are not defined for earlier NonStop 1+ systems or for A06 or earlier NonStop systems.

B00 or later NonStop systems support a larger record length for DP2 key-sequenced files. A NonStop 1+ or pre-B00 NonStop system returns an error for this larger record length.

Files with More than 16 Extents

The DP1 disc process, unlike DP2, limits a file to only 16 extents. On a NonStop 1+ or pre-B00 NonStop system, the FILEINFO procedure returns an allocated-extents value of 16 for any DP2 file having 16 or more extents.

Similarly, if <file> has more than 16 extents and the FUP command INFO <file>, EXTENTS is used on a NonStop 1+ or pre-B00 NonStop system, the command returns information on only the first 16 extents and returns an allocated-extents value of 16 for any DP2 file having 16 or more extents.

An error is returned if the CONTROL procedure on a NonStop 1+ or pre-B00 NonStop system attempts to allocate a 17th (or greater) extent.

Partitioned and Alternate-Key Files

All parts of a file must be under the same type of disc process. That is, the partitions of a file cannot be mixed among DP1 and DP2 volumes, although they can be on different systems.

Similarly, related alternate-key files cannot be on mixed sets of volumes.

The OPEN and CREATE procedures on B00 or later NonStop systems check for these restrictions, but older NonStop and NonStop 1+ systems do not.

Other Considerations

- Although a NonStop system can have a mixture of DP1 and DP2 volumes, the Transaction Monitoring Facility (TMF) requires that all audit trails and audited files in the same system must be under the same kind of disc process.

- The NonStop 1+ system has no special handling for DP2 files. Therefore, it treats remote DP2 files exactly as DP1 files.

- A NonStop 1+ or pre-B00 NonStop system receives error 49 ("access violation") if it attempts to open a partitioned, relative, DP2 file. All other DP2 files are accessible from all file systems.

- A NonStop 1+ or pre-B00 NonStop system receives error 49 ("access violation") if it attempts to gain unstructured access to a structured DP2 file. B00 or later NonStop systems can gain such access, however.

  Because of the different structured block formats, an application that opens structured files for unstructured access under one type of disc process may not work correctly with the other type of disc process.

- Some SETMODE and SETMODEMOWAIT operations apply to DP2 files only. A B00 or later Nonstop system issues these SETMODE requests to DP2 disc processes only, but a NonStop 1+ or pre-B00 NonStop system sends the request to the file's disc process regardless of type. A DP1 disc process receiving such a request rejects it with error 2 ("invalid operation").

INDEX

READER COMMENT CARD

Tandem welcomes your comments on the quality and usefulness of its software documentation. Does this manual serve your needs? If not, how could we improve it? Your comments will be forwarded to the writer for review and action, as appropriate.

If your answer to any of the questions below is "no," please supply detailed information, including page numbers, under Comments. Use additional sheets if necessary.

- ▶ Is this manual technically accurate? Yes ☐ No ☐
- ▶ Is information missing? Yes ☐ No ☐
- ▶ Are the organization and content clear? Yes ☐ No ☐
- ▶ Are the format and packaging convenient? Yes ☐ No ☐

Comments

Name Date

Company

Address

City/State Zip

ENSCRIBE™ Programming Manual

NonStop™ Systems
NonStop 1+™ System

82583 A00

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

B U S I N E S S R E P L Y M A I L

FIRST CLASS PERMIT NO. 482 CUPERTINO, CA, U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

Tandem Computers Incorporated
Attn: Manager—Software Publications
Location 01, Department 6350
19333 Vallco Parkway
Cupertino CA 95014-9990

TAPE

TAPE

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014-2599